User's Manual

スマート・コンフィグレータ

ユーザーズマニュアル　RL78 API リファレンス編

ルネサスマイクロコンピュータ
RL78 ファミリ

ルネサス エレクトロニクス
www.renesas.com

Rev.1.07　2026.1

ご注意書き

1. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。回路、ソフトウェアおよびこれらに関連する情報を使用する場合、お客様の責任において、お客様の機器・システムを設計ください。これらの使用に起因して生じた損害（お客様または第三者いずれに生じた損害も含みます。以下同じです。）に関し、当社は、一切その責任を負いません。
2. 当社製品または本資料に記載された製品データ、図、表、プログラム、アルゴリズム、応用回路例等の情報の使用に起因して発生した第三者の特許権、著作権その他の知的財産権に対する侵害またはこれらに関する紛争について、当社は、何らの保証を行うものではなく、また責任を負うものではありません。
3. 当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
4. 当社製品を組み込んだ製品の輸出入、製造、販売、利用、配布その他の行為を行うにあたり、第三者保有の技術の利用に関するライセンスが必要となる場合、当該ライセンス取得の判断および取得はお客様の責任において行ってください。
5. 当社製品を、全部または一部を問わず、改造、改変、複製、リバースエンジニアリング、その他、不適切に使用しないでください。かかる改造、改変、複製、リバースエンジニアリング等により生じた損害に関し、当社は、一切その責任を負いません。
6. 当社は、当社製品の品質水準を「標準水準」および「高品質水準」に分類しており、各品質水準は、以下に示す用途に製品が使用されることを意図しております。

   標準水準：　コンピュータ、OA 機器、通信機器、計測機器、AV 機器、家電、工作機械、パーソナル機器、産業用ロボット等

   高品質水準：輸送機器（自動車、電車、船舶等）、交通制御（信号）、大規模通信機器、金融端末基幹システム、各種安全制御装置等

   当社製品は、データシート等により高信頼性、Harsh environment 向け製品と定義しているものを除き、直接生命・身体に危害を及ぼす可能性のある機器・システム（生命維持装置、人体に埋め込み使用するもの等）、もしくは多大な物的損害を発生させるおそれのある機器・システム（宇宙機器と、海底中継器、原子力制御システム、航空機制御システム、プラント基幹システム、軍事機器等）に使用されることを意図しておらず、これらの用途に使用することは想定していません。たとえ、当社が想定していない用途に当社製品を使用したことにより損害が生じても、当社は一切その責任を負いません。
7. あらゆる半導体製品は、外部攻撃からの安全性を 100%保証されているわけではありません。当社ハードウェア／ソフトウェア製品にはセキュリティ対策が組み込まれているものもありますが、これによって、当社は、セキュリティ脆弱性または侵害（当社製品または当社製品が使用されているシステムに対する不正アクセス・不正使用を含みますが、これに限りません。）から生じる責任を負うものではありません。当社は、当社製品または当社製品が使用されたあらゆるシステムが、不正な改変、攻撃、ウイルス、干渉、ハッキング、データの破壊または窃盗その他の不正な侵入行為（「脆弱性問題」といいます。）によって影響を受けないことを保証しません。当社は、脆弱性問題に起因しまたはこれに関連して生じた損害について、一切責任を負いません。また、法令において認められる限りにおいて、本資料および当社ハードウェア／ソフトウェア製品について、商品性および特定目的との合致に関する保証ならびに第三者の権利を侵害しないことの保証を含め、明示または黙示のいかなる保証も行いません。
8. 当社製品をご使用の際は、最新の製品情報（データシート、ユーザーズマニュアル、アプリケーションノート、信頼性ハンドブックに記載の「半導体デバイスの使用上の一般的な注意事項」等）をご確認の上、当社が指定する最大定格、動作電源電圧範囲、放熱特性、実装条件その他指定条件の範囲内でご使用ください。指定条件の範囲を超えて当社製品をご使用された場合の故障、誤動作の不具合および事故につきましては、当社は、一切その責任を負いません。
9. 当社は、当社製品の品質および信頼性の向上に努めていますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は、データシート等において高信頼性、Harsh environment 向け製品と定義しているものを除き、耐放射線設計を行っておりません。仮に当社製品の故障または誤動作が生じた場合であっても、人身事故、火災事故その他社会的損害等を生じさせないよう、お客様の責任において、冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、お客様の機器・システムとしての出荷保証を行ってください。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様の機器・システムとしての安全検証をお客様の責任で行ってください。
10. 当社製品の環境適合性等の詳細につきましては、製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制する RoHS 指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。かかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
11. 当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器・システムに使用することはできません。当社製品および技術を輸出、販売または移転等する場合は、「外国為替及び外国貿易法」その他日本国および適用される外国の輸出管理関連法規を遵守し、それらの定めるところに従い必要な手続きを行ってください。
12. お客様が当社製品を第三者に転売等される場合には、事前に当該第三者に対して、本ご注意書き記載の諸条件を通知する責任を負うものといたします。
13. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを禁じます。
14. 本資料に記載されている内容または当社製品についてご不明な点がございましたら、当社の営業担当者までお問合せください。

注 1. 本資料において使用されている「当社」とは、ルネサス エレクトロニクス株式会社およびルネサス エレクトロニクス株式会社が直接的、間接的に支配する会社をいいます。

注 2. 本資料において使用されている「当社製品」とは、注 1 において定義された当社の開発、製造製品をいいます。

(Rev.5.0-1　2020.10)

本社所在地

〒135-0061 東京都江東区豊洲 3-2-24（豊洲フォレシア）

www.renesas.com

お問合せ窓口

弊社の製品や技術、ドキュメントの最新情報、最寄の営業お問合せ窓口に関する情報などは、弊社ウェブサイトをご覧ください。

www.renesas.com/contact/

商標について

ルネサスおよびルネサスロゴはルネサス エレクトロニクス株式会社の商標です。すべての商標および登録商標は、それぞれの所有者に帰属します。

製品ご使用上の注意事項

ここでは、マイコン製品全体に適用する「使用上の注意事項」について説明します。個別の使用上の注意事項については、本ドキュメントおよびテクニカルアップデートを参照してください。

1. 静電気対策
   CMOS 製品の取り扱いの際は静電気防止を心がけてください。CMOS 製品は強い静電気によってゲート絶縁破壊を生じることがあります。運搬や保存の際には、当社が出荷梱包に使用している導電性のトレーやマガジンケース、導電性の緩衝材、金属ケースなどを利用し、組み立て工程にはアースを施してください。プラスチック板上に放置したり、端子を触ったりしないでください。また、CMOS 製品を実装したボードについても同様の扱いをしてください。
2. 電源投入時の処置
   電源投入時は、製品の状態は不定です。電源投入時には、LSI の内部回路の状態は不確定であり、レジスタの設定や各端子の状態は不定です。外部リセット端子でリセットする製品の場合、電源投入からリセットが有効になるまでの期間、端子の状態は保証できません。同様に、内蔵パワーオンリセット機能を使用してリセットする製品の場合、電源投入からリセットのかかる一定電圧に達するまでの期間、端子の状態は保証できません。
3. 電源オフ時における入力信号
   当該製品の電源がオフ状態のときに、入力信号や入出力プルアップ電源を入れないでください。入力信号や入出力プルアップ電源からの電流注入により、誤動作を引き起こしたり、異常電流が流れ内部素子を劣化させたりする場合があります。資料中に「電源オフ時における入力信号」についての記載のある製品は、その内容を守ってください。
4. 未使用端子の処理
   未使用端子は、「未使用端子の処理」に従って処理してください。CMOS 製品の入力端子のインピーダンスは、一般に、ハイインピーダンスとなっています。未使用端子を開放状態で動作させると、誘導現象により、LSI 周辺のノイズが印加され、LSI 内部で貫通電流が流れたり、入力信号と認識されて誤動作を起こす恐れがあります。
5. クロックについて
   リセット時は、クロックが安定した後、リセットを解除してください。プログラム実行中のクロック切り替え時は、切り替え先クロックが安定した後に切り替えてください。リセット時、外部発振子（または外部発振回路）を用いたクロックで動作を開始するシステムでは、クロックが十分安定した後、リセットを解除してください。また、プログラムの途中で外部発振子（または外部発振回路）を用いたクロックに切り替える場合は、切り替え先のクロックが十分安定してから切り替えてください。
6. 入力端子の印加波形
   入力ノイズや反射波による波形歪みは誤動作の原因になりますので注意してください。CMOS 製品の入力がノイズなどに起因して、V_{IL}（Max.）から V_{IH}（Min.）までの領域にとどまるような場合は、誤動作を引き起こす恐れがあります。入力レベルが固定の場合はもちろん、V_{IL}（Max.）から V_{IH}（Min.）までの領域を通過する遷移期間中にチャタリングノイズなどが入らないように使用してください。
7. リザーブアドレス（予約領域）のアクセス禁止
   リザーブアドレス（予約領域）のアクセスを禁止します。アドレス領域には、将来の拡張機能用に割り付けられている リザーブアドレス（予約領域）があります。これらのアドレスをアクセスしたときの動作については、保証できませんので、アクセスしないようにしてください。
8. 製品間の相違について
   型名の異なる製品に変更する場合は、製品型名ごとにシステム評価試験を実施してください。同じグループのマイコンでも型名が違うと、フラッシュメモリ、レイアウトパターンの相違などにより、電気的特性の範囲で、特性値、動作マージン、ノイズ耐量、ノイズ幅射量などが異なる場合があります。型名が違う製品に変更する場合は、個々の製品ごとにシステム評価試験を実施してください。

このマニュアルの使い方

対象者　　このマニュアルは、スマート・コンフィグレータのドライバコード生成の機能を理解し、それを用いたアプリケーション・システムを開発するユーザを対象としています。

目的　　このマニュアルは、スマート・コンフィグレータのドライバコード生成の持つソフトウェア機能をユーザに理解していただき、これを使用するシステムのハードウェア、ソフトウェア開発の参照用資料として役立つことを目的としています。

構成　　このマニュアルは、大きく分けて次の内容で構成しています。

読み方　　このマニュアルを読むにあたっては、電気，論理回路，マイクロコンピュータに関する一般知識が必要となります。

凡例			
	データ表記の重み	:	左が上位桁，右が下位桁
	アクティブ・ロウの表記	:	$\overline{\text{XXX}}$　（端子，信号名称に上線）
	注	:	本文中につけた注の説明
	注意	:	気をつけて読んでいただきたい内容
	備考	:	本文中の補足説明
	数の表記	:	10 進数　...　XXXX
			16 進数　...　0xXXXX

目次

1. 概　説

本章では、スマート・コンフィグレータのドライバコード生成機能（以下、コード生成機能と略します）の概要を示します。

1.1 概　要

スマート・コンフィグレータは、GUI ベースでプロジェクトに必要な各種情報を設定することにより、デバイスが提供している周辺モジュール（クロック発生回路，電圧検出回路など）を制御するソースコード（デバイス・ドライバ・プログラム：C ソース・ファイル、ヘッダ・ファイル）を出力することができます。

1.2 特　長

以下に、スマート・コンフィグレータの特長を示します。

- コード生成
  GUI ベースで設定した情報に応じたデバイス・ドライバ・ファイルを出力するだけでなく、main 関数呼び出しを含んだサンプル・プログラムなど、ビルド環境のプログラム一式を出力することもできます。

- レポート機能
  設定した情報を各種形式のファイルで出力し、設計資料として利用することができます。

- リネーム機能
  出力するフォルダ名、ファイル名、およびソースコードに含まれている API 関数の関数名については、デフォルトの名前が付与されますが、ユーザ独自の名前に変更することもできます。

- ユーザコード保護機能
  各 API 関数には、ユーザが独自にコードを追加できるように、ユーザコード記述用のコメントが設けられています。

  [ユーザコード記述用のコメント]
  /* Start user code for xxxx. Do not edit comment generated here */
  /* End user code. Do not edit comment generated here */
  “xxxx” は追加できるユーザコードによって次のように異なります。
  - “global” — グローバル変数とグローバル関数を追加可能
  - “function” — 関数宣言を.h ファイルに追加可能
  - “user init” — 初期化コードを追加可能
  - 割り込み関数名 — サービス・ルーチン・コードを追加可能
  - “adding” — 関数を.c ファイルに追加可能
  - “include” — インクルードファイルを.c ファイルに追加可能
  - “pragma” — pragma 宣言を.c ファイルに追加可能

  このコメント内にコードを記述すると、再度コード生成した場合でもユーザが記述したコードは保護されます

1.3 注意事項

以下に、スマート・コンフィグレータの注意事項を示します。

- OSS (オープンソースソフトウェア)
  コード生成ツールは OSS を使用しません。

2. 出力ファイル

本章では、スマート・コンフィグレータが出力するファイルについて説明します。

2.1 説 明

以下に、スマート・コンフィグレータが出力するファイルの一覧を示します。

表 2.1 出力ファイル（1/22）

コンポーネント	ファイル名	API 関数名
共通	{project name}.c	main
	r_smc_entry.h	―
	r_cg_systeminit.c	R_Systeminit
	r_cg_macrodriver.h	―
	r_cg_userdefine.h	―
	r_cg_interrupt_handlers.h	―
	r_cg_inthandler.c	―
	r_cg_vect_table.c	―
	r_cg_linker_script.ld	―
	r_cg_port.h	―
	r_cg_pclbuz.h	―
	r_cg_kr.h	―
	r_cg_wdt.h	―
	r_cg_intc.h	―
	r_cg_sms.h	―
	r_cg_elc.h	―
	r_cg_dtc_common.c	R_DTC_Set_PowerOn R_DTC_Set_PowerOff
	r_cg_dtc_common.h	―
	r_cg_dtc.h	―
	r_cg_tau_common.c	R_TAU*m*_Create R_TAU*m*_Set_PowerOn R_TAU*m*_Set_PowerOff R_TAU*m*_Set_Reset R_TAU*m*_Release_Reset
	r_cg_tau_common.h	―
	r_cg_tau.h	―
	r_cg_itl_common.c	R_ITL_Create R_ITL_Start_Interrupt R_ITL_Stop Interrupt R_ITL_Set_PowerOn R_ITL_Set_PowerOff R_ITL_Set_Reset R_ITL_Release_Reset
	r_cg_itl_common_user.c	r_itl_interrupt
	r_cg_itl_common.h	―
	r_cg_itl.h	―

表 2.2 出力ファイル（2/22）

コンポーネント	ファイル名	API 関数名
共通	r_cg_trd_common.c	R_TRD_Create R_TRD_Set_PowerOn R_TRD_Set_PowerOff R_TRD_Set_Reset R_TRD_Release_Reset R_PWMOPA_Set_PowerOn R_PWMOPA_Set_PowerOff R_PWMOPA_Set_Reset R_PWMOPA_Release_Reset R_TRD_ForcedOutput_Enable R_TRD_ForcedOutput_Disable
	r_cg_trd_common.h	―
	r_cg_trd.h	―
	r_cg_trj_common.c	R_TRJ_Set_PowerOn R_TRJ_Set_PowerOff R_TRJ_Set_Reset R_TRJ_Release_Reset
	r_cg_trj_common.h	―
	r_cg_trj.h	―
	r_cg_trg_common.c	R_TRG_Set_PowerOn R_TRG_Set_PowerOff R_TRG_Set_Reset R_TRG_Release_Reset
	r_cg_trg_common.h	―
	r_cg_trg.h	―
	r_cg_trx_common.c	R_TRX_Set_PowerOn R_TRX_Set_PowerOff R_TRX_Set_Reset R_TRX_Release_Reset
	r_cg_trx_common.h	―
	r_cg_trx.h	―
	r_cg_tkb_common.c	R_TKB_Create R_TKB_Set_PowerOn R_TKB_Set_PowerOff R_TKB_Set_Reset R_TKB_Release_Reset
	r_cg_tkb_common.h	―
	r_cg_tkb	―
	r_cg_rtc_common.c	R_RTC_Set_PowerOn R_RTC_Set_PowerOff
	r_cg_rtc_common.h	―
	r_cg_rtc.h	―

表 2.3 出力ファイル（3/22）

コンポーネント	ファイル名	API 関数名
共通	r_cg_it_common.c	R_IT_Set_PowerOn R_IT_Set_PowerOff
	r_cg_it_common.h	―
	r_cg_it.h	―
	r_cg_ad_common.c	R_ADC_Set_PowerOn R_ADC_Set_PowerOff R_ADC_Set_Reset R_ADC_Release_Reset
	r_cg_ad_common.h	―
	r_cg_ad.h	―
	r_cg_da_common.c	R_DAC_Create R_DAC_Set_PowerOn R_DAC_Set_PowerOff R_DAC_Set_Reset R_DAC_Release_Reset
	r_cg_da_common.h	―
	r_cg_da.h	―
	r_cg_comp_common.c	R_COMP_Create R_COMP_Set_PowerOn R_COMP_Set_PowerOff R_COMP_Set_Reset R_COMP_Release_Reset
	r_cg_comp_common.h	―
	r_cg_comp.h	―
	r_cg_pgacomp_common.c	R_PGACOMP_Create R_PGACOMP_Set_PowerOn R_PGACOMP_Set_PowerOff R_PGACOMP_Set_Reset R_PGACOMP_Release_Reset
	r_cg_pgacomp_common.h	―
	r_cg_pgacomp.h	―
	r_cg_sau_common.c	R_SAU*m*_Create R_SAU*m*_Set_PowerOn R_SAU*m*_Set_PowerOff R_SAU*m*_Set_Reset R_SAU*m*_Release_Reset R_SAU*m*_Set_SnoozeOn R_SAU*m*_Set_SnoozeOff
	r_cg_sau_common.h	―
	r_cg_sau.h	―

表 2.4 出力ファイル（4/22）

コンポーネント	ファイル名	API 関数名
共通	r_cg_uarta_common.c	R_UARTA_Create R_UARTA_Set_PowerOn R_UARTA_Set_PowerOff
	r_cg_uarta_common.h	―
	r_cg_uarta.h	―
	r_cg_iica_common.c	R_IICA*n*_Set_PowerOn R_IICA*n*_Set_PowerOff R_IICA*n*_Set_Reset R_IICA*n*_Release_Reset
	r_cg_iica_common.h	―
	r_cg_iica.h	―
	r_cg_rlin3_common.c	R_RLIN3n_Set_PowerOn R_RLIN3n_Set_PowerOff
	r_cg_rlin3_common.h	
	r_cg_rlin3.h	
	r_cg_dali_common.c	R_DALI_Set_PowerOn R_DALI_Set_PowerOff R_DALI_Set_Reset R_DALI_Release_Reset
	r_cg_dali_common.h	
	r_cg_dali.h	
	r_cg_lvd_common.c	R_LVD_Start_Interrupt R_LVD_Stop_Interrupt
	r_cg_lvd_common_user.c	r_lvd_interrupt
	r_cg_lvd_common.h	―
	r_cg_lvd.h	―
	r_cg_remc_common.c	R_REMC_Set_PowerOn R_REMC_Set_PowerOff R_REMC_Set_Reset R_REMC_Release_Reset
	r_cg_remc_common.h	―
	r_cg_remc.h	―
	r_cg_it8bit_common.c	R_IT*m*_Create R_IT*m*_Set_PowerOn R_IT*m*_Set_PowerOff
	r_cg_it8bit_common.h	―
	r_cg_it8bit.h	―
	r_cg_lcd.h	―
	r_cg_osd_common.c	R_OSD_Set_PowerOn R_OSD_Set_PowerOff R_OSD_Set_Reset R_OSD_Release_Reset
	r_cg_osd_common.h	―
	r_cg_osd.h	―

表 2.5 出力ファイル（5/22）

コンポーネント	ファイル名	API 関数名
共通	r_cg_exsd_common.c	R_EXSD_Set_PowerOn R_EXSD_Set_PowerOff R_EXSD_Set_Reset R_EXSD_Release_Reset
	r_cg_exsd_common.h	―
	r_cg_exsd.h	―
ポート機能	{Config_PORT}.c	R_{Config_PORT}_Create R_{Config_PORT}_ReadPmnValues R_{Config_PORT}_ReadDigitalOutputLevel
	{Config_PORT}_user.c	R_{Config_PORT}_Create_UserInit
	{Config_PORT}.h	―
ディレイ・カウンタ	{Config_TAU*m_n*}.c	R_{Config_TAU*m_n*}_Create R_{Config_TAU*m_n*}_Start R_{Config_TAU*m_n*}_Stop R_{Config_TAU*m_n*}_Lower8bits_Start R_{Config_TAU*m_n*}_Lower8bits_Stop R_{Config_TAU*m_n*}_Set_SoftwareTriggerOn
	{Config_TAU*m_n*}_user.c	R_{Config_TAU*m_n*}_Create_UserInit r_{Config_TAU*m_n*}_ interrupt
	{Config_TAU*m_n*}.h	―
分周機能	{Config_TAU*m_n*}.c	R_{Config_TAU*m_n*}_Create R_{Config_TAU*m_n*}_Start R_{Config_TAU*m_n*}_Stop
	{Config_TAU*m_n*}_user.c	R_{Config_TAU*m_n*}_Create_UserInit r_{Config_TAU*m_n*}_ interrupt
	{Config_TAU*m_n*}.h	―
外部イベント・カウンタ（タイマ・アレイ・ユニット）	{Config_TAU*m_n*}.c	R_{Config_TAU*m_n*}_Create R_{Config_TAU*m_n*}_Start R_{Config_TAU*m_n*}_Stop R_{Config_TAUm_n}_Lower8bits_Start R_{Config_TAUm_n}_Lower8bits_Stop
	{Config_TAU*m_n*}_user.c	R_{Config_TAU*m_n*}_Create_UserInit r_{Config_TAU*m_n*}_ interrupt
	{Config_TAU*m_n*}.h	―
外部イベント・カウンタ（タイマ RJ）	{Config_TRJ*n*}.c	R_{Config_TRJ*n*}_Create R_{Config_TRJ*n*}_Start R_{Config_TRJ*n*}_Stop
	{Config_TRJ*n*}_user.c	R_{Config_TRJ*n*}_Create_UserInit r_{Config_TRJ*n*}_interrupt
	{Config_TRJ*n*}.h	―

表 2.6 出力ファイル（6/22）

コンポーネント	ファイル名	API 関数名
入力信号のハイ／ロウ・レベル幅測定（タイマ・アレイ・ユニット）	{Config_TAU*m*_*n*}.c	R_{Config_TAUm_*n*}_Create R_{Config_TAUm_*n*}_Start R_{Config_TAUm_*n*}_Stop R_{Config_TAUm_*n*}_Get_PulseWidth
	{Config_TAU*m*_*n*}_user.c	R_{Config_TAU*m*_*n*}_Create_UserInit r_{Config_TAU*m*_*n*}_ interrupt
	{Config_TAU*m*_*n*}.h	－
入力信号のハイ／ロウ・レベル幅測定（タイマ RJ）	{Config_TRJ*n*}.c	R_{Config_TRJ*n*}_Create R_{Config_TRJ*n*}_Start R_{Config_TRJ*n*}_Stop R_{Config_TRJ*n*}_Get_PulseWidth
	{Config_TRJ*n*}_user.c	R_{Config_TRJ*n*}_Create_UserInit r_{Config_TRJ*n*}_interrupt
	{Config_TRJ*n*}.h	－
PWM 出力（タイマ・アレイ・ユニット）	{Config_TAU*m*_*n*}.c	R_{Config_TAUm_*n*}_Create R_{Config_TAUm_*n*}_Start R_{Config_TAUm_*n*}_Stop
	{Config_TAU*m*_*n*}_user.c	R_{Config_TAU*m*_*n*}_Create_UserInit r_{Config_TAU*m*_*n*}_channel*n*_interrupt r_{Config_TAU*m*_*n*}_channel*p*_interrupt
	{Config_TAU*m*_*n*}.h	－
PWM 出力（PWM モード（リモコン・キャリア波形）を使用したタイマ・アレイ・ユニット）	{Config_TAU0_*m*_TAU0_*n*}.c	R_{Config_TAU0_*m*_TAU0_*n*}_Create R_{Config_TAU0_*m*_TAU0_*n*}_Start R_{Config_TAU0_*m*_TAU0_*n*}_Stop
	{Config_TAU0_*m*_TAU0_*n*}_user.c	R_{Config_TAU0_*m*_TAU0_*n*}_Create_UserInit r_{Config_TAU0_*m*_TAU0_*n*}_channel*m*_interrupt r_{Config_TAU0_*m*_TAU0_*n*}_channel*p*_interrupt r_{Config_TAU0_*m*_TAU0_*n*}_channel*n*_interrupt r_{Config_TAU0_*m*_TAU0_*n*}_channel*q*_interrupt
	{Config_TAU0_*m*_TAU0_*n*}.h	－
PWM 出力（PWM モード／拡張 PWM モードを使用したタイマ RD*n*）	{Config_TRD*n*}.c	R_{Config_TRD*n*}_Create R_{Config_TRD*n*}_Start R_{Config_TRD*n*}_Stop R_{Config_TRD*n*}_Set_TRD*n*_ReloadTrigger
	{Config_TRD*n*}_user.c	R_{Config_TRD*n*}_Create_UserInit r_{Config_TRD*n*}_trd*n*_interrupt
	{Config_TRD*n*}.h	－

表 2.7 出力ファイル（7/22）

コンポーネント	ファイル名	API 関数名
PWM 出力（PWM モード／PWM3 モード／拡張 PWM モード／タイマ KB3 PWM 出力ゲートモードを使用したタイマ RD0 と RD1）	{Config_TRD0_TRD1}.c	R_{Config_TRD0_TRD1}_Create R_{Config_TRD0_TRD1}_Start R_{Config_TRD0_TRD1}_Stop R_{Config_TRD0_TRD1}_Set_TRD*n*_ReloadTrigger R_{Config_TRD0_TRD1}_Set_TRD0_ReloadTrigger R_{Config_TRD0_TRD1}_Set_TRD1_ReloadTrigger
	{Config_TRD0_TRD1}_user.c	R_{Config_TRD0_TRD1}_Create_UserInit r_{Config_TRD0_TRD1}_trd*n*_interruptr
	{Config_TRD0_TRD1}.h	－
PWM 出力（PWM モード／PWM2 モードを使用したタイマ RG）	{Config_TRG}.c	R_{Config_TRG}_Create R_{Config_TRG}_Start R_{Config_TRG}_Stop
	{Config_TRG}_user.c	R_{Config_TRG}_Create_UserInit r_{Config_TRG}_interrupt
	{Config_TRG}.h	－
PWM 出力（単体動作モード（TKBCRn0 レジスタによる周期制御）／単体動作モード（外部トリガ入力による周期制御）／インターリーブ PFC 出力モードを使用したタイマ KB）	{Config_TKB*n*}.c	R_{Config_TKB*n*}_Create R_{Config_TKB*n*}_Start R_{Config_TKB*n*}_Stop R_{Config_TKB*n*}_Set_BatchOverwriteRequestOn R_{Config_TKB*n*}_TKBO*n*0_Forced_Output_Stop_Function1_Start R_{Config_TKB*n*}_TKBO*n*0_Forced_Output_Stop_Function1_Stop R_{Config_TKB*n*}_TKBO*n*1_Forced_Output_Stop_Function1_Start R_{Config_TKB*n*}_TKBO*n*1_Forced_Output_Stop_Function1_Stop R_{Config_TKB*n*}_TKBO*n*0_SmoothStartFunction_Start R_{Config_TKB*n*}_TKBO*n*0_SmoothStartFunction_Stop R_{Config_TKB*n*}_TKBO*n*1_SmoothStartFunction_Start R_{Config_TKB*n*}_TKBO*n*1_SmoothStartFunction_Stop
	{Config_TKB*n*}_user.c	R_{Config_TKB*n*}_Create_UserInit r_{Config_TKB*n*}_ terminated0_interrupt r_{Config_TKB*n*}_ terminated1_interrupt r_{Config_TKB*n*}_ activated0_interrupt r_{Config_TKB*n*}_ activated1_interrupt r_{Config_TKB*n*}_end_count_interrupt
	{Config_TKB*n*}.h	－

表 2.8 出力ファイル（8/22）

コンポーネント	ファイル名	API 関数名
PWM 出力（同時スタート／ストップ・モード（TKBCRn0 レジスタによる周期制御）／同時スタート／ストップ・モード（外部トリガ入力による周期制御）／同時スタート／ストップ・モード（マスタによる周期制御）を使用したタイマ KB）（1 スレーブ）	{Config_TKB0_TKB*n*}.c	R_{Config_TKB0_TKB*n*}_Create R_{Config_TKB0_TKB*n*}_Start R_{Config_TKB0_TKB*n*}_Stop R_{Config_TKB0_TKB*n*}_TKB*m*_Set_BatchOverwriteRequestOn R_{Config_TKB0_TKB*n*}_TKBO*m*0_Forced_Output_Stop_Function1_Start R_{Config_TKB0_TKB*n*}_TKBO*m*0_Forced_Output_Stop_Function1_Stop R_{Config_TKB0_TKB*n*}_TKBO*m*1_Forced_Output_Stop_Function1_Start R_{Config_TKB0_TKB*n*}_TKBO*m*1_Forced_Output_Stop_Function1_Stop R_{Config_TKB0_TKB*n*}_TKBO*m*0_SmoothStartFunction_Start R_{Config_TKB0_TKB*n*}_TKBO*m*0_SmoothStartFunction_Stop R_{Config_TKB0_TKB*n*}_TKBO*m*1_SmoothStartFunction_Start R_{Config_TKB0_TKB*n*}_TKBO*m*1_SmoothStartFunction_Stop
	{Config_TKB0_TKB*n*}_user.c	R_{Config_TKB0_TKB*n*}_Create_UserInit r_{Config_TKB0_TKB*n*}_tkb*m*_terminated0_interrupt r_{Config_TKB0_TKB*n*}_tkb*m*_ terminated1_interrupt r_{Config_TKB0_TKB*n*}_tkb*m*_ activated0_interrupt r_{Config_TKB0_TKB*n*}_tkb*m*_ activated1_interrupt r_{Config_TKB0_TKB*n*}_tkb*m*_end_count_interrupt
	{Config_TKB0_TKB*n*}.h	－

表 2.9 出力ファイル（9/22）

コンポーネント	ファイル名	API 関数名
PWM 出力（同時スタート／ストップ・モード（TKBCRn0 レジスタによる周期制御）／同時スタート／ストップ・モード（外部トリガ入力による周期制御）／同時スタート／ストップ・モード（マスタによる周期制御）を使用したタイマ KB）（2 スレーブ）	{Config_TKB0_TKB1_TKB2}.c	R_{Config_TKB0_TKB1_TKB2}_Create R_{Config_TKB0_TKB1_TKB2}_Start R_{Config_TKB0_TKB1_TKB2}_Stop R_{Config_TKB0_TKB1_TKB2}_TKB*n*_Set_BatchOverwriteRequestOn R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_Forced_Output_Stop_Function1_Start R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_Forced_Output_Stop_Function1_Stop R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_Forced_Output_Stop_Function1_Start R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_Forced_Output_Stop_Function1_Stop R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_SmoothStartFunction_Start R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_SmoothStartFunction_Stop R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_SmoothStartFunction_Start R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_SmoothStartFunction_Stop
	{Config_TKB0_TKB1_TKB2}_user.c	R_{Config_TKB0_TKB1_TKB2}_Create_UserInit r_{Config_TKB0_TKB1_TKB2}_tkb*n*_terminated0_interrupt r_{Config_TKB0_TKB1_TKB2}_tkb*n*_terminated1_interrupt r_{Config_TKB0_TKB1_TKB2}_tkb*n*_ activated0_interrupt r_{Config_TKB0_TKB1_TKB2}_tkb*n*_ activated1_interrupt r_{Config_TKB0_TKB1_TKB2}_tkb*n*_end_count_interrupt
	{Config_TKB0_TKB1_TKB2}.h	―

表 2.10 出力ファイル（10/22）

コンポーネント	ファイル名	API 関数名
入力パルス間隔／周期測定（タイマ・アレイ・ユニット）	{Config_TAUm_n}.c	R_{Config_TAUm_n}_Create R_{Config_TAUm_n}_Start R_{Config_TAUm_n}_Stop R_{Config_TRJn}_Get_PulseWidth
	{Config_TAUm_n}_user.c	R_{Config_TAUm_n}_Create_UserInit r_{Config_TAUm_n}_interrupt
	{Config_TAUm_n}.h	-
入力パルス間隔／周期測定（タイマ RJ）	{Config_TRJn}.c	R_{Config_TRJn}_Create R_{Config_TRJn}_Start R_{Config_TRJn}_Stop R_{Config_TAUm_n}_Get_PulseWidth
	{Config_TRJn}_user.c	R_{Config_TRJn}_Create_UserInit r_{Config_TRJn}_interrupt
	{Config_TRJn}.h	
インターバル・タイマ(タイマ・アレイ・ユニット)	{Config_TAU*m_n*}.c	R_{Config_TAU*m_n*}_Create R_{Config_TAU*m_n*}_Start R_{Config_TAU*m_n*}_Stop R_{Config_TAU*m_n*}_Higher8bits_Start R_{Config_TAU*m_n*}_Higher8bits_Stop R_{Config_TAU*m_n*}_Lower8bits_Start R_{Config_TAU*m_n*}_Lower8bits_Stop
	{Config_TAU*m_n*}_user.c	R_{Config_TAU*m_n*}_Create_UserInit r_{Config_TAU*m_n*}_interrupt r_{Config_TAU*m_n*}_higher8bits_interrupt
	{Config_TAU*m_n*}.h	―
インターバル・タイマ(タイマ RJ)	{Config_TRJ*n*}.c	R_{Config_TRJ*n*}_Create R_{Config_TRJ*n*}_Start R_{Config_TRJ*n*}_Stop
	{Config_TRJ*n*}_user.c	R_{Config_TRJ*n*}_Create_UserInit r_{Config_TRJ*n*}_interrupt
	{Config_TRJ*n*}.h	―
インターバル・タイマ(12 ビット・インターバル・タイマ)	{Config_IT}.c	R_{Config_IT}_Create R_{Config_IT}_Start R_{Config_IT}_Stop
	{Config_IT}_user.c	R_{Config_IT}_Create_UserInit r_{Config_IT}_interrupt
	{Config_IT}.h	―

表 2.11 出力ファイル（11/22）

ワンショット・パルス出力	{Config_TAU*m_n*}.c	R_{Config_TAU*m_n*}_Create R_{Config_TAU*m_n*}_Start R_{Config_TAU*m_n*}_Stop R_{Config_TAU*m_n*}_Set_SoftwareTriggerOn R_{Config_TAU*m_n*}_Set_Get_PulseWidth
	{Config_TAU*m_n*}_user.c	R_{Config_TAU*m_n*}_Create_UserInit r_{Config_TAU*m_n*}_channel*n*_interrupt r_{Config_TAU*m_n*}_channel*p*_interrupt
	{Config_TAU*m_n*}.h	－
方形波出力（タイマ・アレイ・ユニット）	{Config_TAU*m_n*}.c	R_{Config_TAU*m_n*}_Create R_{Config_TAU*m_n*}_Start R_{Config_TAU*m_n*}_Stop R_{Config_TAU*m_n*}_Lower8bits_Start R_{Config_TAU*m_n*}_Lower8bits_Stop
	{Config_TAU*m_n*}_user.c	R_{Config_TAU*m_n*}_Create_UserInit r_{Config_TAU*m_n*}_interrupt
	{Config_TAU*m_n*}.h	－
方形波出力（タイマRJ）	{Config_TRJ*n*}.c	R_{Config_TRJ*n*}_Create R_{Config_TRJ*n*}_Start R_{Config_TRJ*n*}_Stop
	{Config_TRJ*n*}_user.c	R_{Config_TRJ*n*}_Create_UserInit r_{Config_TRJ*n*}_interrupt
	{Config_TRJ*n*}.h	－
インターバル・タイマ（8 ビット・カウント・モードを使用した 32 ビット・インターバル・タイマ）	{Config_ITL*n*}.c	R_{Config_ITL*n*}_Create R_{Config_ITL*n*}_Start R_{Config_ITL*n*}_Stop R_{Config_ITL*n*}_Set_SoftwareTriggerOn R_{Config_ITL*n*}_Set_OperationMode R_{Config_ITL*n*}_Get_CaptureValue
	{Config_ITL*n*}_user.c	R_{Config_ITL*n*}_Create_UserInit r_{Config_ITL*n*}_Callback_Shared_Interrupt
	{Config_ITL*n*}.h	－
インターバル・タイマ（16 ビット・カウント・モードを使用した 32 ビット・インターバル・タイマ）	{Config_ITL*n*_ITL*m*}.c	R_{Config_ITL*n*_ITL*m*}_Create R_{Config_ITL*n*_ITL*m*}_Start R_{Config_ITL*n*_ITL*m*}_Stop R_{Config_ITL*n*_ITL*m*}_Set_SoftwareTriggerOn R_{Config_ITL*n*_ITL*m*}_Set_OperationMode R_{Config_ITL*n*_ITL*m*}_Get_CaptureValue
	{Config_ITL*n*_ITL*m*}_user.c	R_{Config_ITL*n*_ITL*m*}_Create_UserInit r_{Config_ITL*n*_ITL*m*}_Callback_Shared_Interrupt
	{Config_ITL*n*_ITL*m*}.h	－

表 2.12 出力ファイル（12/22）

コンポーネント	ファイル名	API 関数名
インターバル・タイマ（32 ビット・カウント・モードを使用した 32 ビット・インターバル・タイマ）	{Config_ITL000_ITL001_ITL012_ITL013}.c	R_{Config_ITL000_ITL001_ITL012_ITL013}_Create R_{Config_ITL000_ITL001_ITL012_ITL013}_Start R_{Config_ITL000_ITL001_ITL012_ITL013}_Stop R_{Config_ITL000_ITL001_ITL012_ITL013}_Set_OperationMode
	{Config_ITL000_ITL001_ITL012_ITL013}_user.c	R_{Config_ITL000_ITL001_ITL012_ITL013}_Create_UserInit r_{Config_ITL000_ITL001_ITL012_ITL013}_Callback_Shared_Interrupt
	{Config_ITL000_ITL001_ITL012_ITL013}.h	－
インプットキャプチャ機能（タイマ RD）	{Config_TRD*n*}.c	R_{Config_TRD*n*}_Create R_{Config_TRD*n*}_Start R_{Config_TRD*n*}_Stop R_{Config_TRD*n*}_Get_PulseWidth
	{Config_TRD*n*}_user.c	R_{Config_TRD*n*}_Create_UserInit r_{Config_TRD*n*}_trdn_interrupt
	{Config_TRD*n*}.h	－
インプットキャプチャ機能(タイマ RG)	{Config_TRG}.c	R_{Config_TRG}_Create R_{Config_TRG}_Start R_{Config_TRG}_Stop R_{Config_TRG}_Get_PulseWidth
	{Config_TRG}_user.c	R_{Config_TRG}_Create_UserInit r_{Config_TRG}_interrupt
	{Config_TRG}.h	
インプットキャプチャ機能(タイマ RX)	{Config_TRX}.c	R_{Config_TRX}_Create R_{Config_TRX}_Start R_{Config_TRX}_Stop R_{Config_TRX}_Get_BufferValue
	{Config_TRX}_user.c	R_{Config_TRX}_Create_UserInit r_{Config_TRX}_interrupt
	{Config_TRX}.h	
アウトプットコンペア機能（タイマ RD）	{Config_TRD*n*}.c	R_{Config_TRD*n*}_Create R_{Config_TRD*n*}_Start R_{Config_TRD*n*}_Stop
	{Config_TRD*n*}_user.c	R_{Config_TRD*n*}_Create_UserInit R_{Config_TRD*n*}_trd*n*_interrupt
	{Config_TRD*n*}.h	－

表 2.13 出力ファイル（13/22）

コンポーネント	ファイル名	API 関数名
アウトプットコンペア機能（タイマ RG）	{Config_TRG}.c	R_{Config_TRG}_Create R_{Config_TRG}_Start R_{Config_TRG}_Stop
	{Config_TRG}_user.c	R_{Config_TRG}_Create_UserInit r_{Config_TRG}_interrupt
	{Config_TRG}.h	
三相 PWM 出力（タイマ RD）	{Config_TRD0_TRD1}.c	R_{Config_TRD0_TRD1}_Create R_{Config_TRD0_TRD1}_Start R_{Config_TRD0_TRD1}_Stop R_{Config_TRD0_TRD1}_Set_TRD_ReloadTrigger
	{Config_TRD0_TRD1}_user.c	R_{Config_TRD0_TRD1}_Create_UserInit r_{Config_TRD0_TRD1}_trd0_interrupt r_{Config_TRD0_TRD1}_trd1_interrupt
	{Config_TRD0_TRD1}.h	－
PWM オプション・ユニット A（タイマ RD）	{Config_PWMOPA}.c	R_{Config_PWMOPA }_Create R_{Config_PWMOPA }_Software_Release
	{Config_ PWMOPA }_user.c	R_{Config_PWMOPA }_Create_UserInit
	{Config_PWMOPA}.h	－
位相計数モード	{Config_TRG}.c	R_{Config_TRG}_Create R_{Config_TRG}_Start R_{Config_TRG}_Stop R_{Config_TRG}_Get_MeasurementCapture
	{Config_TRG}_user.c	R_{Config_TRG}_Create_UserInit r_{Config_TRG}_interrupt r_{Config_TRG}_clear_interrupt r_{Config_TRG}_capture_interrupt
	{Config_TRG}.h	
クロック出力／ブザー出力制御回路	{Config_PCLBUZ*n*}.c	R_{Config_PCLBUZ*n*}_Create R_{Config_PCLBUZ*n*}_Start R_{Config_PCLBUZ*n*}_Stop
	{Config_PCLBUZ*n*}_user.c	R_{Config_PCLBUZ*n*}_Create_UserInit
	{Config_PCLBUZ*n*}.h	－

表 2.14 出力ファイル（14/22）

コンポーネント	ファイル名	API 関数名
リアルタイム・クロック	{Config_RTC}.c	R_{Config_RTC}_Create R_{Config_RTC}_Start R_{Config_RTC}_Stop R_{Config_RTC}_Set_HourSystem R_{Config_RTC}_Set_CounterValue R_{Config_RTC}_Get_CounterValue R_{Config_RTC}_Set_ConstPeriodInterruptOn R_{Config_RTC}_Set_ConstPeriodInterruptOff R_{Config_RTC}_Set_AlarmOn R_{Config_RTC}_Set_AlarmOff R_{Config_RTC}_Set_AlarmValue R_{Config_RTC}_Get_AlarmValue R_{Config_RTC}_Set_RTC1HZOn R_{Config_RTC}_Set_RTC1HZOff
	{Config_RTC}_user.c	R_{Config_RTC}_Create_UserInit r_{Config_RTC}_interrupt r_{Config_RTC}_callback_constperiod r_{Config_RTC}_callback_alarm
	{Config_RTC}.h	－
A/D コンバータ	{Config_ADC}.c	R_{Config_ADC}_Create R_{Config_ADC}_Start R_{Config_ADC}_Stop R_{Config_ADC}_Set_OperationOn R_{Config_ADC}_Set_OperationOff R_{Config_ADC}_Set_ADChannel R_{Config_ADC}_ADS*n*_Set_ADChannel R_{Config_ADC}_Set_SnoozeOn R_{Config_ADC}_Set_SnoozeOff R_{Config_ADC}_Set_TestChannel R_{Config_ADC}_Get_Result_10bit R_{Config_ADC}_Get_Result_8bit R_{Config_ADC}_Get_Result_12bit R_{Config_ADC}_ADS*n*_Get_Result_10bit R_{Config_ADC}_ADS*n*_Get_Result_8bit R_{Config_ADC}_ADS*n*_Get_Result_12bit
	{Config_ADC}_user.c	R_{Config_ADC}_Create_UserInit r_{Config_ADC}_interrupt r_{Config_ADC}_ad*n*_interrupt
	{Config_ADC}.h	－

表 2.15 出力ファイル（15/22）

コンポーネント	ファイル名	API 関数名
12 ビット A/D シングル・スキャン	{Config_S12AD*n*}.c	R_{Config_S12AD*n*}_Create R_{Config_S12AD*n*}_Start R_{Config_S12AD*n*}_Stop R_{Config_S12AD*n*}_Get_ValueResult
	{Config_S12AD*n*}_user.c	R_{Config_S12AD*n*}_Create_UserInit r_{Config_S12AD*n*}_interrupt
	{Config_S12AD*n* }.h	−
12 ビット A/D 連続スキャン	{Config_S12AD*n*}.c	R_{Config_S12AD*n*}_Create R_{Config_S12AD*n*}_Start R_{Config_S12AD*n*}_Stop R_{Config_S12AD*n*}_Get_ValueResult
	{Config_S12AD*n*}_user.c	R_{Config_S12AD*n*}_Create_UserInit r_{Config_S12AD*n*}_interrupt
	{Config_S12AD*n* }.h	−
12 ビット A/D グループ・スキャン	{Config_S12AD*n*}.c	R_{Config_S12AD*n*}_Create R_{Config_S12AD*n*}_Start R_{Config_S12AD*n*}_Stop R_{Config_S12AD*n*}_Get_ValueResult
	{Config_S12AD*n*}_user.c	R_{Config_S12AD*n*}_Create_UserInit r_{Config_S12AD*n*}_interrupt r_{Config_S12AD*n*}_groupb_interrupt
	{Config_S12AD*n* }.h	−
D/A コンバータ	{Config_DAC*n*}.c	R_{Config_DAC*n*}_Create R_{Config_DAC*n*}_Start R_{Config_DAC*n*}_Stop R_{Config_DAC*n*}_Set_ConversionValue
	{Config_DAC*n*}_user.c	R_{Config_DAC*n*}_Create_UserInit
	{Config_DAC*n*}.h	−
データ・トランスファ・コントローラ	{Config_DTC}.c	R_{Config_DTC}_Create R_{Config_DTCD*n*}_Start R_{Config_DTCD*n*}_Stop
	{Config_DTC}_user.c	R_{Config_DTC}_Create_UserInit
	{Config_DTC}.h	−
コンパレータ	{Config_COMP*n*}.c	R_{Config_COMP*n*}_Create R_{Config_COMP*n*}_Start R_{Config_COMP*n*}_Stop
	{Config_COMP*n*}_user.c	R_{Config_COMP*n*}_Create_UserInit r_{Config_COMP*n*}_interrupt
	{Config_COMP*n*}.h	−

表 2.16 出力ファイル（16/22）

コンポーネント	ファイル名	API 関数名
プログラマブル・ゲイン・アンプ	{Config_PGA}.c	R_{Config_PGA}_Create R_{Config_PAG}_Start R_{Config_PAG}_Stop
	{Config_PGA}_user.c	R_{Config_PGA}_Create_UserInit
	{Config_PGA}.h	
SPI (CSI) 通信	{Config_CSI*p*}.c	R_{Config_CSI*p*}_Create R_{Config_CSI*p*}_Start R_{Config_CSI*p*}_Stop R_{Config_CSI*p*}_Send R_{Config_CSI*p*}_Receive R_{Config_CSI*p*}_Send_Receive
	{Config_CSI*p*}_user.c	R_{Config_CSI*p*}_Create_UserInit r_{Config_CSI*p*}_interrupt r_{Config_CSI*p*}_callback_sendend r_{Config_CSI*p*}_callback_receiveend r_{Config_CSI*p*}_callback_error
	{Config_CSI*p*}.h	－
UART 通信（シリアル・アレイ・ユニット）	{Config_UART*q*}.c	R_{Config_UART*q*}_Create R_{Config_UART*q*}_Start R_{Config_UART*q*}_Stop R_{Config_UART*q*}_Send R_{Config_UART*q*}_Receive R_{Config_UART*q*}_Loopback_Enable R_{Config_UART*q*}_Loopback_Disable
	{Config_UART*q*}_user.c	R_{Config_UART*q*}_Create_UserInit r_{Config_UART*q*}_interrupt_send r_{Config_UART*q*}_interrupt_receive r_{Config_UART*q*}_interrupt_error r_{Config_UART*q*}_callback_sendend r_{Config_UART*q*}_callback_receiveend r_{Config_UART*q*}_callback_error r_{Config_UART*q*}_callback_softwareoverrun
	{Config_UART*q*}.h	－

表 2.17 出力ファイル（17/22）

コンポーネント	ファイル名	API 関数名
UART 通信（シリアル・インタフェース UARTA）	{Config_UARTA*n*}.c	R_{Config_UARTA*n*}_Create R_{Config_UARTA*n*}_Start R_{Config_UARTA*n*}_Stop R_{Config_UARTA*n*}_Send R_{Config_UARTA*n*}_Receive R_{Config_UARTA*n*}_Loopback_Enable R_{Config_UARTA*n*}_Loopback_Disable
	{Config_UARTA*n*}_user.c	R_{Config_UARTA*n*}_Create_UserInit R_{Config_UARTA*n*}_PollingEnd_UserCode r_{Config_UARTA*n*}_interrupt_send r_{Config_UARTA*n*}_interrupt_receive r_{Config_UARTA*n*}_interrupt_error r_{Config_UARTA*n*}_callback_sendend r_{Config_UARTA*n*}_callback_receiveend r_{Config_UARTA*n*}_callback_error
	{Config_UARTA*n*}.h	－
UART 通信（LIN/UART モジュール）	{Config_RLIN3*n*}.c	R_{Config_RLIN3*n*}_Create R_{Config_RLIN3*n*}_Start R_{Config_RLIN3*n*}_Stop R_{Config_RLIN3*n*}_Send R_{Config_RLIN3*n*}_Receive
	{Config_RLIN3*n*}_user.c	R_{Config_RLIN3*n*}_Create_UserInit r_{Config_RLIN3*n*}_interrupt_send r_{Config_RLIN3*n*}_interrupt_receive r_{Config_RLIN3*n*}_interrupt_error r_{Config_RLIN3*n*}_callback_sendend r_{Config_RLIN3*n*}_callback _receiveend r_{Config_RLIN3*n*}_callback _error
	{Config_RLIN3*n*}.h	－

表 2.18 出力ファイル（18/22）

コンポーネント	ファイル名	API 関数名
DALI 通信（コントロールデバイス）	{Config_DALI}.c	R_{Config_DALI}_Create R_{Config_DALI}_Start R_{Config_DALI}_Stop R_{Config_DALI}_SoftwareReset R_{Config_DALI}_EnableForceActiveState R_{Config_DALI}_DisableForceActiveState R_{Config_DALI}_GetStatus R_{Config_DALI}_Send R_{Config_DALI}_GetReceivedFrame
	{Config_DALI}_user.c	R_{Config_DALI}_Create_UserInit r_{Config_DALI}_interrupt_send r_{Config_DALI}_interrupt_receive r_{Config_DALI}_interrupt_error r_{Config_DALI}_interrupt_falling_edge_detection r_{Config_DALI}_interrupt_power_down_detection r_{Config_DALI}_interrupt_collision_detection r_{Config_DALI}_interrupt_stop_bit_detection r_{Config_DALI}_ callback_sendend r_{Config_DALI}_callback_receiveend r_{Config_DALI}_callback_error
	{Config_DALI}.h	―
DALI 通信（コントロールギア）	{Config_DALI}.c	R_{Config_DALI}_Create R_{Config_DALI}_Start R_{Config_DALI}_Stop R_{Config_DALI}_SoftwareReset R_{Config_DALI}_EnableForceActiveState R_{Config_DALI}_DisableForceActiveState R_{Config_DALI}_GetStatus R_{Config_DALI}_Send R_{Config_DALI}_GetReceivedFrame
	{Config_DALI}_user.c	R_{Config_DALI}_Create_UserInit r_{Config_DALI}_interrupt_error r_{Config_DALI}_interrupt_falling_edge_detection r_{Config_DALI}_interrupt_power_down_detection r_{Config_DALI}_interrupt_stop_bit_detection r_{Config_DALI}_ callback_sendend r_{Config_DALI}_callback_receiveend r_{Config_DALI}_callback_error
	{Config_DALI}.h	―

表 2.19 出力ファイル（19/22）

コンポーネント	ファイル名	API 関数名
IIC 通信(マスタ・モード)（シリアル・アレイ・ユニット）	{Config_IIC*r*}.c	R_{Config_IIC*r*}_Create R_{Config_IIC*r*}_StartCondition R_{Config_IIC*r*}_StopCondition R_{Config_IIC*r*}_Stop R_{Config_IIC*r*}_Master_Send R_{Config_IIC*r*}_Master_Receive
	{Config_IIC*r*}_user.c	R_{Config_IIC*r*}_Create_UserInit r_{Config_IIC*r*}_interrupt r_{Config_IIC*r*}_callback_master_sendend r_{Config_IIC*r*}_callback_master_receiveend r_{Config_IIC*r*}_callback_master_error
	{Config_IIC*r*}.h	―
IIC 通信(マスタ・モード)（シリアル・インタフェース IICA）	{Config_IICA*n*}.c	R_{Config_IICA*n*}_Create R_{Config_IICA*n*}_StopCondition R_{Config_IICA*n*}_Stop R_{Config_IICA*n*}_Master_Send R_{Config_IICA*n*}_Master_Receive R_{Config_IICA*n*}_Check_Comstate R_{Config_IICAn}_Poll R_{Config_IICAn}_Wait_Comend R_{Config_IICA*n*}_Bus_Check R_{Config_IICA*n*}_StartCondition R_{Config_IICA*n*}_Wait_Time
	{Config_IICA*n*}_user.c	R_{Config_IICA*n*}_Create_UserInit r_{Config_IICA*n*}_interrupt r_{Config_IICA*n*}_master_handler r_{Config_IICA*n*}_callback_master_sendend r_{Config_IICA*n*}_callback_master_receiveend r_{Config_IICA*n*}_callback_master_error
	{Config_IICA*n*}.h	―
IIC 通信(マスタ・モード、EEPROM 通信)（シリアル・インタフェース IICA）	{Config_IICA*n*}.c	R_{Config_IICA*n*}_Create R_{Config_IICA*n*}_StopCondition R_{Config_IICA*n*}_Stop R_{Config_IICA*n*}_Master_Send R_{Config_IICA*n*}_Master_Receive R_{Config_IICA*n*}_Check_Comstate R_{Config_IICA*n*}_Poll R_{Config_IICA*n*}_Wait_Comend R_{Config_IICA*n*}_Bus_Check R_{Config_IICA*n*}_StartCondition R_{Config_IICA*n*}_Wait_Time

表 2.20 出力ファイル（20/22）

コンポーネント	ファイル名	API 関数名
IIC 通信(マスタ・モード、EEPROM 通信)（シリアル・インタフェース IICA）	{Config_IICA*n*}_user.c	R_{Config_IICA*n*}_Create_UserInit r_{Config_IICA*n*}_interrupt r_{Config_IICA*n*}_master_handler r_{Config_IICA*n*}_callback_master_sendend r_{Config_IICA*n*}_callback_master_receiveend r_{Config_IICA*n*}_callback_master_error
	{Config_IICA*n*}.h	－
IIC 通信(スレーブ・モード)（シリアル・インタフェース IICA）	{Config_IICA*n*}.c	R_{Config_IICA*n*}_Create R_{Config_IICA*n*}_Stop R_{Config_IICA*n*}_Slave_Send R_{Config_IICA*n*}_Slave_Receive R_{Config_IICA*n*}_Set_WakeupOn R_{Config_IICA*n*}_Set_WakeupOff
	{Config_IICA*n*}_user.c	R_{Config_IICA*n*}_Create_UserInit r_{Config_IICA*n*}_interrupt r_{Config_IICA*n*}_slave_handler r_{Config_IICA*n*}_callback_slave_sendend r_{Config_IICA*n*}_callback_slave_receiveend r_{Config_IICA*n*}_callback_slave_error r_{Config_IICA*n*}_callback_getstopcondition
	{Config_IICA*n*}.h	－
割り込みコントローラ	{Config_INTC}.c	R_{Config_INTC}_Create R_{Config_INTC}_INTP*n*_Start R_{Config_INTC}_INTP*n*_Stop
	{Config_INTC}_user.c	R_{Config_INTC}_Create_UserInit r_{Config_INTC}_intp*n*_interrupt
	{Config_INTC}.h	－
電圧検出回路	{Config_LVD*n*}.c	R_{Config_LVD*n*}_Create R_{Config_LVD*n*} _Start R_{Config_LVD*n*} _Stop
	{Config_LVD*n*}_user.c	R_{Config_LVD*n*}_Create_UserInit
	{Config_LVD*n*}.h	－
SNOOZE モード・シーケンサ	{Config_SMS}.c	R_{Config_SMS}_Create R_{Config_SMS}_Start R_{Config_SMS}_Stop R_{Config_SMS}_GetStatus R_{Config_SMS}_GetReturn R_{Config_SMS}_TriggerWait_Disable R_{Config_SMS}_TriggerWait_Enable R_{Config_SMS}_Set_PowerOn R_{Config_SMS}_Set_PowerOff R_{Config_SMS}_Set_Reset R_{Config_SMS}_Release_Reset

表 2.21 出力ファイル（21/22）

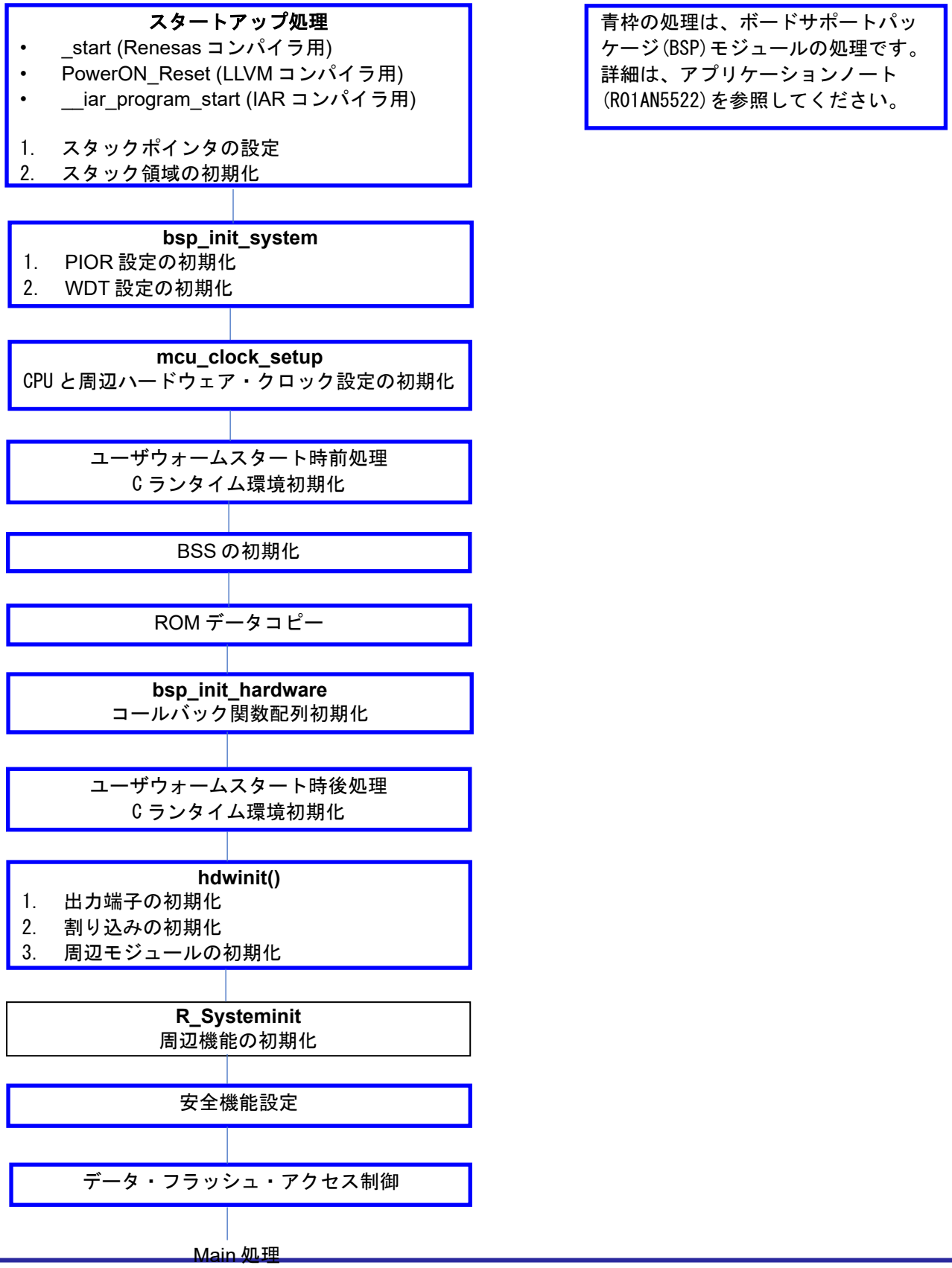
コンポーネント	ファイル名	API 関数名
SNOOZE モード・シーケンサ	{Config_SMS}_user.c	R_{Config_SMS}_Create_UserInit r_{Config_SMS}_interrupt
	{Config_SMS}.h	―
キー割り込み	{Config_KR}.c	R_{Config_KR}_Create R_{Config_KR} _Start R_{Config_KR} _Stop
	{Config_KR}_user.c	R_{Config_KR}_Create_UserInit r_{Config_KR}_interrupt
	{Config_KR}.h	―
リモコン信号受信機能	{Config_REMC}.c	R_{Config_REMC}_Create R_{Config_REMC} _Start R_{Config_REMC} _Stop R_{Config_REMC}_Read
	{Config_REMC}_user.c	R_{Config_REMC}_Create_UserInit r_{Config_REMC}_interrupt r_{Config_REMC}_callback_receiveend r_{Config_REMC}_callback_comparematch r_{Config_REMC}_callback_receiveerror r_{Config_REMC}_callback_bufferfull r_{Config_REMC}_callback_header r_{Config_REMC}_callback_data0 r_{Config_REMC}_callback_data1 r_{Config_REMC}_callback_specialdata
	{Config_REMC}.h	―
ウォッチドッグ・タイマ	{Config_WDT}.c	R_{Config_WDT}_Create R_{Config_WDT} _Restart
	{Config_WDT}_user.c	R_{Config_WDT}_Create_UserInit r_{Config_WDT}_interrupt
	{Config_WDT}.h	―
ロジック&イベント・リンク・コントローラ (ELCL)	Config_*xxx*}.c	R_{Config_*xxx*}_Create R_{Config_*xxx*}_Start R_{Config_*xxx*}_Stop
	{Config_*xxx*}_user.c	R_{Config_*xxx*}_Create_UserInit r_{Config_*xxx*}_interrupt
	{Config_*xxx*}.h	―
イベント・リンク・コントローラ	Config_ELC_*xxx*}.c	R_{Config_ELC}_Create R_{Config_ELC}_Stop
	{Config_ELC_*xxx*}_user.c	R_{Config_ELC}_Create_UserInit
	{Config_ELC_*xxx*}.h	―

表 2.22 出力ファイル（22/22）

コンポーネント	ファイル名	API 関数名
インターバル・タイマ（8 ビット・カウント・モードを使用した 8 ビット・インターバル・タイマ）	{Config_IT*mn*}.c	R_{Config_IT*mn*}_Create R_{Config_IT*mn*}_Start R_{Config_IT*mn*}_Stop
	{Config_IT*mn*}_user.c	R_{Config_IT*mn*}_Create_UserInit r_{Config_IT*mn*}_interrupt
	{Config_IT*mn*}.h	−
インターバル・タイマ(16 ビット・カウンタ・モードを使用した 8 ビット・インターバル・タイマ)	{Config_IT*m*0_IT*m*1}.c	R_{Config_IT*m*0_IT*m*1}_Create R_{Config_IT*m*0_IT*m*1}_Start R_{Config_IT*m*0_IT*m*1}_Stop
	{Config_IT*m*0_IT*m*1}_user.c	R_{Config_IT*m*0_IT*m*1}_Create_UserInit r_{Config_IT*m*0_IT*m*1}_interrupt
	{Config_IT*m*0_IT*m*1}.h	−
LCD コントローラ／ドライバ	{Config_LCD}.c	R_{Config_LCD}_Create R_{Config_LCD}_Start R_{Config_LCD}_Stop R_{Config_LCD}_Voltage_On R_{Config_LCD}_Voltage_Off R_{Config_LCD}_Set_DisplayData
	{Config_LCD}_user.c	R_{Config_LCD}_Create_UserInit
	{Config_LCD}.h	−
発振停止検出回路	{Config_OSD}.c	R_{Config_OSD}_Create R_{Config_OSD }_Start R_{Config_OSD}_Stop
	{Config_OSD}_user.c	R_{Config_OSD}_Create_UserInit r_{Config_OSD}_interrupt
	{Config_OSD}.h	−
外部サンプリング	{Config_EXSD}.c	R_{Config_EXSD}_Create R_{Config_EXSD }_Start R_{Config_EXSD}_Stop
	{Config_EXSD}_user.c	R_{Config_EXSD}_Create_UserInit r_{Config_EXSD}_interrupt
	{Config_EXSD}.h	−

3. 初期化

本章では、コード生成が出力するファイルによる初期化の流れについて説明します。



4. API 関数

本章では、スマート・コンフィグレータが出力する API 関数について説明します。

4.1 概　　要

以下に、スマート・コンフィグレータが API 関数を出力する際の命名規則を示します。

- マクロ名
  すべて大文字。
  なお、先頭に“ 数字 ”が付与されている場合、該当数字（16 進数値）とマクロ値は同値。

- ローカル変数名
  すべて小文字。

- グローバル変数名
  先頭に “g” を付与し、構成単語の先頭のみ大文字。

- グローバル変数へのポインタ名
  先頭に “gp” を付与し、構成単語の先頭のみ大文字。

- 列挙指定子 enum の要素名
  すべて大文字。

備考　スマート・コンフィグレータが生成するコードには、レジスタの反映待ち処理等で for 文、while 文、do while 文（ループ処理）を使用している関数があります。無限ループに対するフェイルセーフ処理が必要な場合は、生成されたコードを確認の上、処理を追加してください。

4.2 関数リファレンス

本節では、スマート・コンフィグレータが出力する API 関数について、次の記述フォーマットに従って説明します。

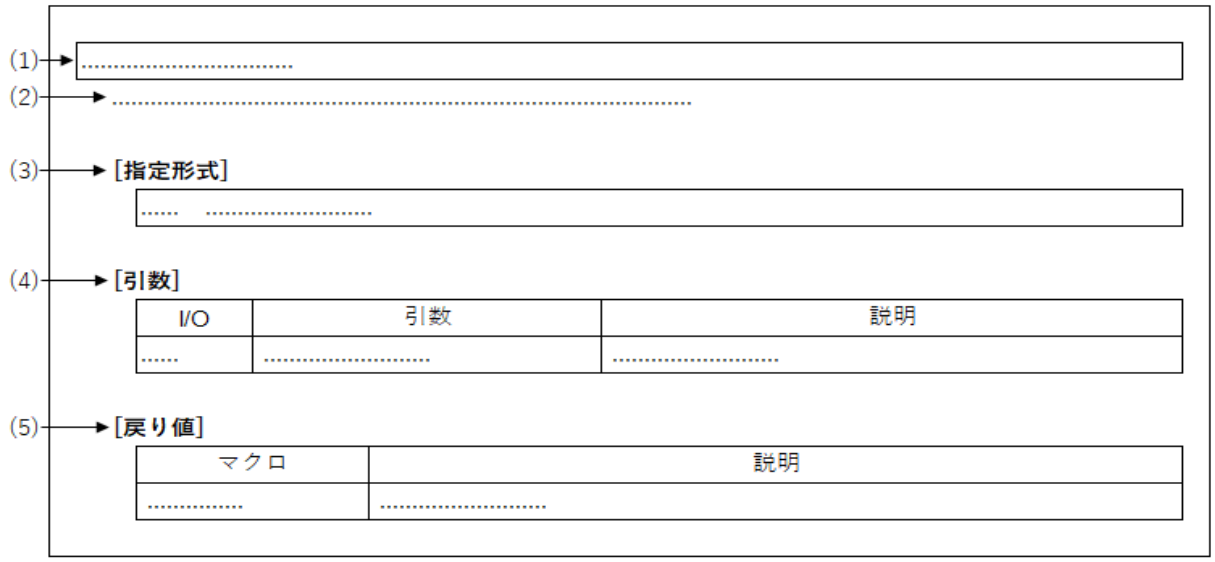


図 4.1 API 関数の記述フォーマット

(1) 名称
API 関数の名称を示しています。

(2) 機能概要
API 関数の機能概要を示しています。

(3) [指定形式]
API 関数を C 言語で呼び出す際の記述形式を示しています。

(4) [引数]
API 関数の引数を次の形式で示しています。

I/O	引数	説明
(a)	(b)	(c)

(a) I/O
引数の種類
I … 入力引数
O … 出力引数

(b) 引数
引数のデータタイプ

(c) 説明
引数の説明

(5) [戻り値]
API 関数からの戻り値を次の形式で示しています。

マクロ	説明
(a)	(b)

(a) マクロ
戻り値のマクロ

(b) 説明
戻り値の説明

4.2.1 共通

以下に、スマート・コンフィグレータが共通用として出力する API 関数の一覧を示します。

表 4.1 共通用 API 関数（1/4）

API 関数名	周辺機能	機能概要
main	-	main 関数です。
R_Systeminit	-	各種周辺機能を制御するうえで必要となる初期化処理を行います。
R_DTC_Set_PowerOn	データ・トランスファ・コントローラ	DTC へのクロック供給を開始します。
R_DTC_Set_PowerOff		DTC へのクロック供給を停止します。
R_TAU*m*_Create	タイマ・アレイ・ユニット	TAU*m* を制御する前に必要な初期化処理を実行します (TAU*m* への入力クロックの供給を許可し、TAU*m* モジュールを初期化します）。
R_TAU*m*_Set_PowerOn		TAU*m* へのクロック供給を開始します。
R_TAU*m*_Set_PowerOff		TAU*m* へのクロック供給を停止します。
R_TAU*m*_Set_Reset		TAU*m* モジュールをリセット状態にします。
R_TAUm_Release_Reset		TAU*m* モジュールのリセット状態を解除します。
R_ITL_Create	32 ビット・インターバル・タイマ	32 ビット・インターバル・タイマを制御する前に必要な初期化処理を実行します (入力クロック供給を有効にし、ITL*m* モジュールを初期化します）。
R_ITL_Start_Interrupt		INTITL 割り込みを開始します。
R_ITL_Stop Interrupt		INTITL 割り込みを停止します。
R_ITL_Set_PowerOn		32 ビット・インターバル・タイマへのクロック供給を開始します。
R_ITL_Set_PowerOff		32 ビット・インターバル・タイマへのクロック供給を停止します。
R_ITL_Set_Reset		32 ビット・インターバル・タイマ・モジュールをリセット状態にします。
R_ITL_Release_Reset		32 ビット・インターバル・タイマ・モジュールのリセット状態を解除します。
r_itl_interrupt		32 ビット・インターバル・タイマ割り込み（INTITL）に応じて処理を実行します。
R_TRD_Create	タイマ RD	TRD を制御する前に必要な初期化処理を実行します (TRD 入力クロック供給を有効にし、TRD モジュールを初期化します)。
R_TRD_Set_PowerOn		TRD へのクロック供給を開始します。
R_TRD_Set_PowerOff		TRD へのクロック供給を停止します。
R_TRD_Set_Reset		TRD・モジュールをリセット状態にします。
R_TRD_Release_Reset		TRD・モジュールのリセット状態を解除します。
R_PWMOPA_Set_PowerOn		PWMOPA へのクロック供給を開始します。
R_PWMOPA_Set_PowerOff		PWMOPA へのクロック供給を停止します。
R_PWMOPA_Set_Reset		PWMOPA・モジュールをリセット状態にします。
R_PWMOPA_Release_Reset		PWMOPA・モジュールのリセット状態を解除します。

表 4.2 共通用 API 関数（2/4）

API 関数名	周辺機能	機能概要
R_TRD_ ForcedOutput_Enable	タイマ RD	TRD パルス出力の強制遮断を有効にします。
R_TRD_ ForcedOutput_Disable		TRD パルス出力の強制遮断を無効にします。
R_TRJ_Set_PowerOn	タイマ RJ	TRJ へのクロック供給を開始します。
R_TRJ_Set_PowerOff		TRJ へのクロック供給を停止します。
R_TRJ_Set_Reset		TRJ・モジュールをリセット状態にします。
R_TRJ_Release_Reset		TRJ・モジュールのリセット状態を解除します。
R_TRG_Set_PowerOn	タイマ RG	TRG へのクロック供給を開始します。
R_TRG_Set_PowerOff		TRG へのクロック供給を停止します。
R_TRG_Set_Reset		TRG・モジュールをリセット状態にします。
R_TRG_Release_Reset		TRG・モジュールのリセット状態を解除します。
R_TRX_Set_PowerOn	タイマ RX	TRX へのクロック供給を開始します。
R_TRX_Set_PowerOff		TRX へのクロック供給を停止します。
R_TRX_Set_Reset		TRX・モジュールをリセット状態にします。
R_TRX_Release_Reset		TRX・モジュールのリセット状態を解除します。
R_TKB_Create	タイマ KB	TKB モジュールを制御する前に必要な初期化処理を実行します（入力クロック供給を有効にし、TKB モジュールを初期化します）。
R_TKB_Set_PowerOn		TKB へのクロック供給を開始します。
R_TKB_Set_PowerOff		TKB へのクロック供給を停止します。
R_TKB_Set_Reset		TKB・モジュールをリセット状態にします。
R_TKB_Release_Reset		TKB・モジュールのリセット状態を解除します。
R_RTC_Set_PowerOn	リアルタイム・クロック	RTC へのクロック供給を開始します。
R_RTC_Set_PowerOff		RTC へのクロック供給を停止します。
R_IT_Set_PowerOn	12 ビット・インターバル・タイマ	12 ビット・インターバル・タイマのクロック供給を開始します。
R_IT_Set_PowerOff		12 ビット・インターバル・タイマのクロック供給を停止します。
R_ADC_Set_PowerOn	A/D コンバータ	A/D コンバータへのクロック供給を開始します。
R_ADC_Set_PowerOff		A/D コンバータへのクロック供給を停止します。
R_ADC_Set_Reset		A/D コンバータ・モジュールをリセット状態にします。
R_ADC_Release_Reset		A/D コンバータ・モジュールのリセット状態を解除します。
R_DAC_Create	D/A コンバータ	DAC モジュールを制御する前に必要な初期化処理を実行します(入力クロック供給を有効にし、DA*m* モジュールを初期化します)。
R_DAC_Set_PowerOn		D/A コンバータへのクロック供給を開始します。
R_DAC_Set_PowerOff		D/A コンバータへのクロック供給を停止します。
R_DAC_Set_Reset		D/A コンバータ・モジュールをリセット状態にします。
R_DAC_Release_Reset		D/A コンバータ・モジュールのリセット状態を解除します。

表 4.3 共通用 API 関数（3/4）

API 関数名	周辺機能	機能概要
R_COMP_Create	コンパレータ	コンパレータ・モジュールを制御する前に必要な初期化処理を実行します（入力クロック供給を有効にし、COMPm モジュールを初期化します）。
R_COMP_Set_PowerOn		コンパレータへのクロック供給を開始します。
R_COMP_Set_PowerOff		コンパレータへのクロック供給を停止します。
R_COMP_Set_Reset		コンパレータ・モジュールをリセット状態にします。
R_COMP_Release_Reset		コンパレータ・モジュールのリセット状態を解除します。
R_PGACOMP_Create	コンパレータとプログラマブル・ゲイン・アンプ	コンパレータとプログラマブル・ゲイン・アンプ・モジュールを制御する前に必要な初期化処理を実行します（入力クロック供給を有効にし、COMPm モジュールを初期化します）。
R_PGACOMP_Set_PowerOn		コンパレータとプログラマブル・ゲイン・アンプへのクロック供給を開始します。
R_PGACOMP_Set_PowerOff		コンパレータとプログラマブル・ゲイン・アンプへのクロック供給を停止します。
R_PGACOMP_Set_Reset		コンパレータとプログラマブル・ゲイン・アンプ・モジュールをリセット状態にします。
R_PGACOMP_Release_Reset		コンパレータとプログラマブル・ゲイン・アンプ・モジュールのリセット状態を解除します。
R_SAU*m*_Create	シリアル・アレイ・ユニット	SAUm を制御する前に必要な初期化処理を実行します（入力クロック供給を有効にし、SAUm モジュールを初期化します）。
R_SAU*m*_Set_PowerOn		SAUm へのクロック供給を開始します。
R_SAU*m*_Set_PowerOff		SAUm へのクロック供給を停止します。
R_SAU*m*_Set_Reset		SAUm モジュールをリセット状態にします。
R_SAU*m*_Release_Reset		SAUm モジュールのリセット状態を解除します。
R_SAU*m*_Set_SnoozeOn		SAUm のウエイクアップ機能を許可します。
R_SAU*m*_Set_SnoozeOff		SAUm のウエイクアップ機能を禁止します。
R_UARTA_Create	シリアル・インタフェース UARTA	UARTA0/UARTA1 を制御する前に必要な初期化処理を実行します（入力クロック供給を有効にし、モジュールを初期化します）。
R_UARTA_Set_PowerOn		UARTA0/UARTA1 へのクロック供給を開始します。
R_UARTA_Set_PowerOff		UARTA0/UARTA1 へのクロック供給を停止します。
R_IICA*n*_Set_PowerOn	シリアル・インタフェース IICA	IICAn へのクロック供給を開始します。
R_IICA*n*_Set_PowerOff		IICAn へのクロック供給を停止します。
R_IICA*n*_Set_Reset		IICAn モジュールをリセット状態にします。
R_IICA*n*_Release_Reset		IICAn モジュールのリセット状態を解除します。
R_RLIN3*n*_Set_PowerOn	LIN/UART module	RLIN3*n* へのクロック供給を開始します。
R_RLIN3*n*_Set_PowerOff		RLIN3*n* へのクロック供給を停止します。

表 4.4 共通用 API 関数（4/4）

API 関数名	周辺機能	機能概要
R_DALI_Set_PowerOn	デジタル調光照明インタフェース(DALI)	DALI へのクロック供給を開始します。
R_DALI_Set_PowerOff		DALI へのクロック供給を停止します。
R_DALI_Set_Reset		DALI モジュールをリセット状態にします。
R_DALI_Release_Reset		DALI モジュールのリセット状態を解除します。
R_LVD_Start_Interrupt	電圧検出回路	INTLVI 割り込みを開始します。
R_LVD_Stop_Interrupt		INTLVI 割り込みを停止します。
r_lvd_interrupt		INTLVI 割り込みに応じて処理を実行します。
R_REMC_Set_PowerOn	リモコン信号受信機能	REMC へのクロック供給を開始します。
R_REMC_Set_PowerOff		REMC へのクロック供給を停止します。
R_REMC_Set_Reset		REMC モジュールをリセット状態にします。
R_REMC_Release_Reset		REMC モジュールのリセット状態を解除します。
R_IT*m*_Create	8 ビット・インターバル・タイマ	IT*m* を制御する前に必要な初期化処理を実行します(IT*m* 入力クロック供給を有効にし、IT*m* モジュールを初期化します）。
R_IT*m*_Set_PowerOn		IT*m* へのクロック供給を開始します。
R_IT*m*_Set_PowerOff		IT*m* へのクロック供給を停止します。
R_OSD_Set_PowerOn	発振停止検出回路	OSD へのクロック供給を開始します。
R_OSD_Set_PowerOff		OSD へのクロック供給を停止します。
R_OSD_Set_Reset		OSD モジュールをリセット状態にします。
R_OSD_Release_Reset		OSD モジュールのリセット状態を解除します。
R_EXSD_Set_PowerOn	外部サンプリング	EXSD へのクロック供給を開始します。
R_EXSD_Set_PowerOff		EXSD へのクロック供給を停止します。
R_EXSD_Set_Reset		EXSD モジュールをリセット状態にします。
R_EXSD_Release_Reset		EXSD モジュールのリセット状態を解除します。

main

main 関数です。

備考　単体版および CS+で使用する場合、main()に以下のコードを手動で追加することに注意下さい。

1) #include “r_smc_entry.h” の追加
2) EI() の追加

[指定形式]

```
void    main(void);
```

[引数]

なし

[戻り値]

なし

R_Systeminit

各種周辺機能を制御するうえで必要となる初期化処理を行います。

[指定形式]

```
void    R_Systeminit(void);
```

[引数]

なし

[戻り値]

なし

R_DTC_Set_PowerOn

DTC へのクロック供給を開始します。

[指定形式]

```
void    R_DTC_Set_PowerOn(void);
```

[引数]

なし

[戻り値]

なし

R_DTC_Set_PowerOff

DTC へのクロック供給を停止します。

[指定形式]

```
void    R_DTC_Set_PowerOff(void);
```

[引数]

なし

[戻り値]

なし

R_TAU*m*_Create

TAU*m* を制御する前に必要な初期化処理を実行します（TAU*m* への入力クロックの供給を許可し、TAU*m* モジュールを初期化します）。

備考　この API 関数は、main()関数が実行される前に R_Systeminit から呼び出されます。

［指定形式］

```
void    R_TAUm_Create(void);
```

備考　*m* はユニット番号を示します。

［引数］

なし

［戻り値］

なし

R_TAU*m*_Set_PowerOn

TAU*m* へのクロック供給を開始します。

[指定形式]

```
void    R_TAUm_ Set_PowerOn(void);
```

備考　　*m* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

R_TAU*m*_Set_PowerOff

TAU*m* へのクロック供給を停止します。

[指定形式]

void R_TAU*m*_ Set_PowerOff(void);

備考 *m* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

R_TAU*m*_Set_Reset

TAU*m* モジュールをリセット状態にします。

[指定形式]

void R_TAU*m*_ Set_Reset(void);

備考 *m* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

R_TAU*m*_Release_Reset

TAU*m* モジュールのリセット状態を解除します。

[指定形式]

```
void    R_TAUm_Release_Reset(void);
```

備考　　*m* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

R_ITL_Create

32 ビット・インターバル・タイマを制御する前に必要な初期化処理を実行します（入力クロック供給を有効にし、ITL*m* モジュールを初期化します）。

備考　この API 関数は、main()関数が実行される前に R_Systeminit から呼び出されます。

[指定形式]

```
void    R_ITL_Create(void);
```

[引数]

なし

[戻り値]

なし

R_ITL_Start_Interrupt

INTITL 割り込みを開始します。

備考　この API 関数を呼び出すことで、32 ビット・インターバル・タイマ割り込みが有効になります。32 ビット・インターバル・タイマ割り込みを使用する場合は、この API 関数を R_{Config_ITL000_ITL001_ITL012_ITL013}_Start、または R_{Config_ITL*n*_ITL*m*}_Start、または R_{Config_ITL*n*}_Start と共に呼び出してください。

[指定形式]

```
void    R_ITL_Start_Interrupt(void);
```

[引数]

なし

[戻り値]

なし

R_ITL_Stop_Interrupt

INTITL 割り込みを停止します。

[指定形式]

```
void    R_ITL_Stop_Interrupt(void);
```

[引数]

なし

[戻り値]

なし

R_ITL_Set_PowerOn

32 ビット・インターバル・タイマへのクロック供給を開始します。

[指定形式]

```
void    R_ITL_Set_PowerOn(void);
```

[引数]

なし

[戻り値]

なし

R_ITL_Set_PowerOff

32 ビット・インターバル・タイマへのクロック供給を停止します。

[指定形式]

```
void    R_ITL_Set_PowerOff(void);
```

[引数]

なし

[戻り値]

なし

R_ITL_Set_Reset

32 ビット・インターバル・タイマ・モジュールをリセット状態にします。

[指定形式]

```
void    R_ITL_Set_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_ITL_Release_Reset

32 ビット・インターバル・タイマ・モジュールのリセット状態を解除します。

[指定形式]

```
void    R_ITL_Release_Reset(void);
```

[引数]

なし

[戻り値]

なし

r_itl_interrupt

32 ビット・インターバル・タイマ割り込み（INTITL）の発生により処理を実行します。

備考　この API 関数は、INTITL の割り込みハンドラとして呼び出され、チャネル 0～3 のいずれかのカウンタ値が比較値と一致したときに発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_itl_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_itl_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_itl_interrupt(void);
```

[引数]

なし

[戻り値]

なし

R_TRD_Create

TRD を制御する前に必要な初期化処理を実行します（TRD 入力クロック供給を有効にし、TRD モジュールを初期化します）。

備考　　この API 関数は、main()関数が実行される前に R_Systeminit から呼び出されます。

[指定形式]

```
void    R_TRD_Create(void);
```

[引数]

なし

[戻り値]

なし

R_TRD_Set_PowerOn

TRD へのクロック供給を開始します。

[指定形式]

```
void    R_TRD_ Set_PowerOn(void);
```

[引数]

なし

[戻り値]

なし

R_TRD_Set_PowerOff

TRD へのクロック供給を停止します。

[指定形式]

```
void    R_TRD_ Set_PowerOff(void);
```

[引数]

なし

[戻り値]

なし

R_TRD_Set_Reset

TRD・モジュールをリセット状態にします。

[指定形式]

```
void    R_TRD_ Set_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_TRD_Release_Reset

TRD・モジュールのリセット状態を解除します。

[指定形式]

```
void    R_TRD_ RIease_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_PWMOPA_Set_PowerOn

PWMOPA へのクロック供給を開始します。

[指定形式]

```
void    R_PWMOPA_ Set_PowerOn(void);
```

[引数]

なし

[戻り値]

なし

R_PWMOPA_Set_PowerOff

PWMOPA へのクロック供給を停止します。

[指定形式]

```
void    R_PWMOPA_ Set_PowerOff(void);
```

[引数]

なし

[戻り値]

なし

R_PWMOPA_Set_Reset

PWMOPA・モジュールをリセット状態にします。

[指定形式]

```
void    R_PWMOPA_ Set_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_PWMOPA_Release_Reset

PWMOPA・モジュールのリセット状態を解除します。

[指定形式]

```
void    R_PWMOPA_ Release_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_TRD_ ForcedOutput_Enable

TRD パルス出力の強制遮断を有効にします。タイマ・カウンタ実行中は呼び出せません。
R_{Config_TRD*n*}_Start()/R_{Config_TRD0_TRD1}_Start() の前に呼び出してください。

[指定形式]

```
void    R_TRD_ ForcedOutput_Enable(void);
```

[引数]
なし

[戻り値]
なし

R_TRD_ ForcedOutput_Disable

TRD パルス出力の強制遮断を無効にします。タイマ・カウンタ実行中は呼び出せません。
R_{Config_TRD*n*}_Stop()/R_{Config_TRD0_TRD1}_Stop() の後に呼び出してください。

[指定形式]

```
void    R_TRD_ ForcedOutput_Disable (void);
```

[引数]
なし

[戻り値]
なし

R_TRJ_Set_PowerOn

TRJ へのクロック供給を開始します。

[指定形式]

```
void    R_TRJ_ Set_PowerOn(void);
```

[引数]

なし

[戻り値]

なし

R_TRJ_Set_PowerOff

TRJ へのクロック供給を停止します。

[指定形式]

```
void    R_TRJ_ Set_PowerOff(void);
```

[引数]

なし

[戻り値]

なし

R_TRJ_Set_Reset

TRJ・モジュールをリセット状態にします。

[指定形式]

```
void    R_TRJ_ Set_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_TRJ_Release_Reset

TRJ・モジュールのリセット状態を解除します。

[指定形式]

```
void    R_TRJ_ Release_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_TRG_Set_PowerOn

TRG へのクロック供給を開始します。

[指定形式]

```
void    R_TRG_ Set_PowerOn(void);
```

[引数]

なし

[戻り値]

なし

R_TRG_Set_PowerOff

TRG へのクロック供給を停止します。

[指定形式]

```
void    R_TRG_ Set_PowerOff(void);
```

[引数]

なし

[戻り値]

なし

R_TRG_Set_Reset

TRG・モジュールをリセット状態にします。

[指定形式]

```
void    R_TRG_ Set_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_TRG_Release_Reset

TRG・モジュールのリセット状態を解除します。

[指定形式]

```
void    R_TRG_ Release_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_TRX_Set_PowerOn

TRX へのクロック供給を開始します。

[指定形式]

```
void    R_TRX_ Set_PowerOn(void);
```

[引数]

なし

[戻り値]

なし

R_TRX_Set_PowerOff

TRX へのクロック供給を停止します。

[指定形式]

```
void    R_TRX_ Set_PowerOff(void);
```

[引数]

なし

[戻り値]

なし

R_TRX_Set_Reset

TRX・モジュールをリセット状態にします。

[指定形式]

```
void    R_TRX_ Set_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_TRX_Release_Reset

TRX・モジュールのリセット状態を解除します。

[指定形式]

```
void    R_TRX_ Release_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_TKB_Create

TKB モジュールを制御する前に必要な初期化処理を実行します（入力クロック供給を有効にし、TKB モジュールを初期化します）。

備考　この API 関数は、main()関数が実行される前に R_Systeminit から呼び出されます。

[指定形式]

```
void    R_TKB_Create(void);
```

[引数]

なし

[戻り値]

なし

R_TKB_Set_PowerOn

TKB へのクロック供給を開始します。

[指定形式]

```
void    R_TKB_ Set_PowerOn(void);
```

[引数]

なし

[戻り値]

なし

R_TKB_Set_PowerOff

TKB へのクロック供給を停止します。

[指定形式]

```
void    R_TKB_ Set_PowerOff(void);
```

[引数]

なし

[戻り値]

なし

R_TKB_Set_Reset

TKB・モジュールをリセット状態にします。

[指定形式]

```
void    R_TKB_ Set_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_TKB_Release_Reset

TKB・モジュールのリセット状態を解除します。

[指定形式]

```
void    R_TKB_ Release_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_RTC_Set_PowerOn

RTC へのクロック供給を開始します。

[指定形式]

```
void    R_RTC_Set_PowerOn(void);
```

[引数]

なし

[戻り値]

なし

R_RTC_Set_PowerOff

RTC へのクロック供給を停止します。

[指定形式]

```
void    R_RTC_Set_PowerOff(void);
```

[引数]

なし

[戻り値]

なし

R_IT_Set_PowerOn

12 ビット・インターバル・タイマのクロック供給を開始します。

[指定形式]

```
void    R_IT_Set_PowerOn(void);
```

[引数]

なし

[戻り値]

なし

R_IT_Set_PowerOff

12 ビット・インターバル・タイマのクロック供給を停止します。

[指定形式]

```
void    R_IT_Set_PowerOff(void);
```

[引数]

なし

[戻り値]

なし

R_ADC_Set_PowerOn

A/D コンバータへのクロック供給を開始します。

[指定形式]

```
void    R_ADC_Set_PowerOn(void);
```

[引数]

なし

[戻り値]

なし

R_ADC_Set_PowerOff

A/D コンバータへのクロック供給を停止します。

[指定形式]

```
void    R_ADC_Set_PowerOff(void);
```

[引数]

なし

[戻り値]

なし

R_ADC_Set_Reset

A/D コンバータ・モジュールをリセット状態にします。

[指定形式]

```
void    R_ADC_Set_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_ADC_Release_Reset

A/D コンバータ・モジュールのリセット状態を解除します。

[指定形式]

```
void    R_ADC_Release_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_DAC_Create

DAC モジュールを制御する前に必要な初期化処理を実行します（入力クロック供給を有効にし、DA*m* モジュールを初期化します）。

備考　この API 関数は、main()関数が実行される前に R_Systeminit から呼び出されます。

[指定形式]

```
void    R_DAC_Create(void);
```

[引数]

なし

[戻り値]

なし

R_DAC_Set_PowerOn

D/A コンバータへのクロック供給を開始します。

[指定形式]

```
void    R_DAC_Set_PowerOn(void);
```

[引数]

なし

[戻り値]

なし

R_DAC_Set_PowerOff

D/A コンバータへのクロック供給を停止します。

[指定形式]

```
void    R_DAC_Set_PowerOff(void);
```

[引数]

なし

[戻り値]

なし

R_DAC_Set_Reset

D/A コンバータ・モジュールをリセット状態にします。

[指定形式]

```
void    R_DAC_Set_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_DAC_Release_Reset

D/A コンバータ・モジュールのリセット状態を解除します。

[指定形式]

```
void    R_DAC_Release_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_COMP_Create

コンパレータ・モジュールを制御する前に必要な初期化処理を実行します（入力クロック供給を有効にし、COMP*m* モジュールを初期化します）。

備考　この API 関数は、main()関数が実行される前に R_Systeminit から呼び出されます。

[指定形式]

```
void    R_COMP_Create(void);
```

[引数]

なし

[戻り値]

なし

R_COMP_Set_PowerOn

コンパレータへのクロック供給を開始します。

[指定形式]

```
void    R_COMP_Set_PowerOn(void);
```

[引数]

なし

[戻り値]

なし

R_COMP_Set_PowerOff

コンパレータへのクロック供給を停止します。

[指定形式]

```
void    R_COMP_Set_PowerOff(void);
```

[引数]

なし

[戻り値]

なし

R_COMP_Set_Reset

コンパレータ・モジュールをリセット状態にします。

[指定形式]

```
void    R_COMP_Set_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_COMP_Release_Reset

コンパレータ・モジュールのリセット状態を解除します。

[指定形式]

```
void    R_COMP_Release_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_PGACOMP_Create

コンパレータとプログラマブル・ゲイン・アンプ・モジュールを制御する前に必要な初期化処理を実行します（入力クロック供給を有効にし、COMP*m* モジュールを初期化します）。

備考　この API 関数は、main()関数が実行される前に R_Systeminit から呼び出されます。

[指定形式]

```
void    R_PGACOMP_Create(void);
```

[引数]

なし

[戻り値]

なし

R_PGACOMP_Set_PowerOn

コンパレータとプログラマブル・ゲイン・アンプへのクロック供給を開始します。

[指定形式]

```
void    R_PGACOMP_Set_PowerOn(void);
```

[引数]

なし

[戻り値]

なし

R_PGACOMP_Set_PowerOff

コンパレータとプログラマブル・ゲイン・アンプへのクロック供給を停止します。

[指定形式]

```
void    R_PGACOMP_Set_PowerOff(void);
```

[引数]

なし

[戻り値]

なし

R_PGACOMP_Set_Reset

コンパレータとプログラマブル・ゲイン・アンプ・モジュールをリセット状態にします。

[指定形式]

```
void    R_PGACOMP_Set_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_PGACOMP_Release_Reset

コンパレータとプログラマブル・ゲイン・アンプ・モジュールのリセット状態を解除します。

[指定形式]

```
void    R_PGACOMP_Release_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_SAU*m*_Create

SAU*m* を制御する前に必要な初期化処理を実行します（入力クロック供給を有効にし、SAU*m* モジュールを初期化します）。

備考　この API 関数は、main()関数が実行される前に R_Systeminit から呼び出されます。

[指定形式]

```
void    R_SAUm_Create(void);
```

備考　*m* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

R_SAU*m*_Set_PowerOn

SAU*m* へのクロック供給を開始します。

[指定形式]

```
void    R_SAUm_Set_PowerOn(void);
```

備考　　*m* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

R_SAU*m*_Set_PowerOff

SAU*m* へのクロック供給を停止します。

[指定形式]

```
void    R_SAUm_Set_PowerOff(void);
```

備考　　*m* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

R_SAU*m*_Set_Reset

SAU*m* モジュールをリセット状態にします。

[指定形式]

void R_SAU*m*_Set_Reset(void);

備考 *m* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

R_SAU*m*_Release_Reset

SAU*m* モジュールのリセット状態を解除します。

[指定形式]

```
void    R_SAUm_Release_Reset(void);
```

備考　　*m* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

R_SAU*m*_Set_SnoozeOn

SAU*m* のウエイクアップ機能を許可します。

[指定形式]

```
void    R_SAUm_Set_SnoozeOn(void);
```

備考　　*m* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

R_SAU*m*_Set_SnoozeOff

SAU*m* のウエイクアップ機能を禁止します。

[指定形式]

void R_SAU*m*_Set_SnoozeOff(void);

備考 *m* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

R_UARTA_Create

UARTA0/UARTA1 を制御する前に必要な初期化処理を実行します（入力クロック供給を有効にし、モジュールを初期化します）。

備考　この API 関数は、main()関数が実行される前に R_Systeminit から呼び出されます。

[指定形式]

```
void    R_UARTA_Create(void);
```

[引数]

なし

[戻り値]

なし

R_UARTA_Set_PowerOn

UARTA0/UARTA1 へのクロック供給を開始します。

[指定形式]

```
void    R_UARTA_Set_PowerOn(void);
```

[引数]

なし

[戻り値]

なし

R_UARTA_Set_PowerOff

UARTA0/UARTA1 へのクロック供給を停止します。

[指定形式]

```
void    R_UARTA_Set_PowerOff(void);
```

[引数]

なし

[戻り値]

なし

R_IICA*n*_Set_PowerOn

IICA*n* へのクロック供給を開始します。

[指定形式]

```
void    R_IICAn_Set_PowerOn(void);
```

備考　　*n* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

R_IICA*n*_Set_PowerOff

IICA*n* へのクロック供給を停止します。

[指定形式]

```
void    R_IICAn_Set_PowerOff(void);
```

備考　　*n* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

R_IICA*n*_Set_Reset

IICA*n* モジュールをリセット状態にします。

[指定形式]

```
void    R_IICAn_Set_Reset(void);
```

備考　　*n* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

R_IICA*n*_Release_Reset

IICA*n* モジュールのリセット状態を解除します。

[指定形式]

```
void    R_IICAn_Release_Reset(void);
```

備考　　*n* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

R_RLIN3*n*_Set_PowerOn

RLIN3*n* へのクロック供給を開始します。

[指定形式]

```
void    R_RLIN3n_Set_PowerOn(void);
```

備考　　*n* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

R_RLIN3*n*_Set_PowerOff

RLIN3*n* へのクロック供給を停止します。

[指定形式]

```
void    R_RLIN3n_Set_PowerOff(void);
```

備考　　*n* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

R_DALI_Set_PowerOn

DALI へのクロック供給を開始します。

[指定形式]

```
void    R_DALI_Set_PowerOn(void);
```

[引数]

なし

[戻り値]

なし

R_DALI_Set_PowerOff

DALI へのクロック供給を停止します。

[指定形式]

```
void    R_DALI_Set_PowerOff(void);
```

[引数]

なし

[戻り値]

なし

R_DALI_Set_Reset

DALI モジュールをリセット状態にします。

[指定形式]

```
void    R_DALI_Set_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_DALI_Release_Reset

DALI モジュールのリセット状態を解除します。

[指定形式]

```
void    R_DALI_Release_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_LVD_Start_Interrupt

INTLVI 割り込みを開始します。

[指定形式]

```
void    R_LVD_Start_Interrupt(void);
```

[引数]

なし

[戻り値]

なし

R_LVD_Stop_Interrupt

INTLVI 割り込みを停止します。

[指定形式]

```
void    R_LVD_Stop_Interrupt(void);
```

[引数]

なし

[戻り値]

なし

r_lvd_interrupt

INTLVI 割り込みに伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_lvd_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_lvd_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_lvd_interrupt(void);
```

[引数]

なし

[戻り値]

なし

R_REMC_Set_PowerOn

REMC へのクロック供給を開始します。

[指定形式]

```
void    R_REMC_Set_PowerOn(void);
```

[引数]

なし

[戻り値]

なし

R_REMC_Set_PowerOff

REMC へのクロック供給を停止します。

[指定形式]

```
void    R_REMC_Set_PowerOff(void);
```

[引数]

なし

[戻り値]

なし

R_REMC_Set_Reset

REMC モジュールをリセット状態にします。

[指定形式]

```
void    R_REMC_Set_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_REMC_Release_Reset

REMC モジュールのリセット状態を解除します。

[指定形式]

```
void    R_REMC_Release_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_IT*m*_Create

IT*m* を制御する前に必要な初期化処理を実行します（IT*m* 入力クロック供給を有効にし、IT*m* モジュールを初期化します）。

備考　この API 関数は、main()関数が実行される前に R_Systeminit から呼び出されます。

［指定形式］

```
void    R_ITm_Create(void);
```

備考　*m* はユニット番号を示します。

［引数］

なし

［戻り値］

なし

R_ITm_Set_PowerOn

ITm へのクロック供給を開始します。

[指定形式]

```
void    R_ITm_Set_PowerOn(void);
```

備考　　m はユニット番号を示します。

[引数]

なし

[戻り値]

なし

R_IT*m*_Set_PowerOff

IT*m* へのクロック供給を停止します。

[指定形式]

```
void    R_ITm_Set_PowerOff(void);
```

備考　　*m* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

R_OSD_Set_PowerOn

発振停止検出回路へのクロック供給を開始します。

[指定形式]

```
void    R_OSD_Set_PowerOn(void);
```

[引数]

なし

[戻り値]

なし

R_OSD_Set_PowerOff

発振停止検出回路へのクロック供給を停止します。

[指定形式]

```
void    R_OSD_Set_PowerOff(void);
```

[引数]

なし

[戻り値]

なし

R_OSD_Set_Reset

発振停止検出回路モジュールをリセット状態にします。

[指定形式]

```
void    R_OSD_Set_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_OSD_Release_Reset

発振停止検出回路モジュールのリセット状態を解除します。

[指定形式]

```
void    R_OSD_Release_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_EXSD_Set_PowerOn

外部サンプリングへのクロック供給を開始します。

[指定形式]

```
void    R_EXSD_Set_PowerOn(void);
```

[引数]

なし

[戻り値]

なし

R_EXSD_Set_PowerOff

外部サンプリングへのクロック供給を停止します。

[指定形式]

```
void    R_EXSD_Set_PowerOff(void);
```

[引数]

なし

[戻り値]

なし

R_EXSD_Set_Reset

外部サンプリング・モジュールをリセット状態にします。

[指定形式]

```
void    R_EXSD_Set_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_EXSD_Release_Reset

外部サンプリング・モジュールのリセット状態を解除します。

[指定形式]

```
void    R_EXSD_Release_Reset(void);
```

[引数]

なし

[戻り値]

なし

使用例

R_*Xxxx*_Set_PowerOn() 、 R_*Xxxx*_Set_PowerOff() 、 R_*Xxxx*_Set_Reset() 、 R_*Xxxx*_Release_Reset()、R_ *Xxxx*_Start_Interrupt()、R_ *Xxxx*_Stop_Interrupt() の使用例です。

(*Xxxx* は、ペリフェラル名です。次のサンプルコードでは、例として 32 ビット・インターバル・タイマ（ITL）を使用しています)

main.c

```
#include "r_smc_entry.h"

void main(void);

void main(void)
{
    // 割り込みを有効にします。
    EI();

    // ITL の各チャネル間で共有される INTITL 割り込みを有効にします。
    R_ITL_Start_Initerrupt();

    // 各チャネルの ITL を有効にします。
    R_Config_ITL000_Start();
    R_Config_ITL001_Start();

    // 各チャネルの ITL を無効にします。
    R_Config_ITL000_Stop();
    R_Config_ITL001_Stop();

    // ITL の各チャネル間で共有される INTITL 割り込みを無効にします。
    R_ITL_Stop_Initerrupt();

    // ITL が停止すると、消費電力とノイズが低減されます。
    R_Config_ ITL_Set_PowerOff();

    // ITL を再度使用するには、入力クロックを供給します。
    R_Config_ ITL_Set_PowerOn();

    // ITL をリセット状態に設定します。
    R_ ITL_Set_Reset();

    // ITL を再度使用するには、ITL をリセット状態から解除します。
    R_ ITL_ Release_Reset();

    // ITL を再度使用するには、初期化処理を実行します。
    R_ITL_Create();

    // ITL を再度使用するには、各チャネルの ITL を有効にします。
    R_Config_ITL000_Start();

    // ITL を再度使用するには、各チャネルの ITL を無効にします。
    R_Config_ITL000_Stop();

}
```

4.2.2 ポート機能

以下に、スマート・コンフィグレータがポート機能用として出力する API 関数の一覧を示します。

表 4.5 ポート機能用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_PORT}_Create	ポート機能	ポートを制御する前に必要な初期化処理を実行します。
R_{Config_PORT}_ReadPmnValues		端子が出力モードのときに読み込んだポートの出力ラッチの値を指定します。
R_{Config_PORT}_ReadDigitalOutputLevel		端子が出力モードのときに読み込んだポートの出力レベルを指定します。
R_{Config_PORT}_Create_UserInit		ポート機能に関するユーザ独自の初期化処理を実行します。

R_Config_PORT_Create

ポートを制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_PORT}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_PORT}_ReadPmnValues

端子が出力モードのときに読み込んだポートの出力ラッチの値を指定します。

[指定形式]

```
void    R_{Config_PORT}_ReadPmnValues(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_PORT}_ReadDigitalOutputLevel

端子が出力モードのときに読み込んだポートの出力レベルを指定します。

[指定形式]

```
void    R_{Config_PORT}_ReadDigitalOutputLevel(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_PORT}_Create_UserInit

ポート機能に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_PORT}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_PORT}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

使用例

端子が出力モードのときに読み取られるポート端子の出力レベルを設定する例です。

main.c

```
#include "r_smc_entry.h"

void main(void);

void main(void)
{
    EI();    // 割り込みを有効にします。

    // 出力モードでポート端子の出力レベルを読み取ります。
    R_Config_PORT_ ReadDigitalOutputLevel ();
}
```

4.2.3 ディレイ・カウンタ

以下に、スマート・コンフィグレータがディレイ・カウンタ用として出力する API 関数の一覧を示します。

表 4.6 ディレイ・カウンタ用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TAU*m*_*n*}_Create	タイマ・アレイ・ユニット	TAU*m* チャネル・モジュールをディレイ・カウンタ・モードで制御する前に必要な初期化処理を実行します。
R_{Config_TAU*m*_*n*}_Start		TAU*m* のチャネル *n* のカウンタを起動します。
R_{Config_TAU*m*_*n*}_Stop		TAU*m* のチャネル *n* のカウンタを停止します。
R_{Config_TAUm_*n*}_Lower8bits_Start		TAU*m* のチャネル *n* の下位側 8 ビット・カウンタを起動します。
R_{Config_TAUm_*n*}_Lower8bits_Stop		TAU*m* のチャネル *n* の下位側 8 ビット・カウンタを停止します。
R_{Config_TAU*m*_*n*}_Set_SoftwareTriggerOn		ソフトウェア・トリガを発生させます。
R_{Config_TAUm_*n*}_Create_UserInit		TAU*m* のチャネル n に関するユーザ独自の初期化処理を実行します。
r_{Config_TAUm_n}_interrupt		INTTM*mn* 割り込みに伴う処理を実行します。

R_{Config_TAU*m*_*n*}_Create

TAU*m* チャネル・モジュールをディレイ・カウンタ・モードで制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_TAU*m*_Create から呼び出されます。

[指定形式]

```
void    R_{Config_TAUm_n}_Create(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Start

TAU*m* のチャネル *n* のカウンタを起動します。

[指定形式]

```
void    R_{Config_TAUm_n}_Start(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Stop

TAU*m* のチャネル *n* のカウンタを停止します。

[指定形式]

```
void    R_{Config_TAUm_n}_Stop(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Lower8bits_Start

TAU*m* のチャネル *n* の下位側 8 ビット・カウンタを起動します。

[指定形式]

```
void    R_{Config_TAUm_n}_Lower8bits_Start(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Lower8bits_Stop

TAU*m* のチャネル *n* の下位側 8 ビット・カウンタを停止します。

[指定形式]

```
void    R_{Config_TAUm_n}_Lower8bits_Stop(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Set_SoftwareTriggerOn

ソフトウェア・トリガを発生させます。

[指定形式]

```
void    R_{Config_TAUm_n}_Set_SoftwareTriggerOn(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Create_UserInit

TAU*m* のチャネル *n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TAU*m*_*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TAUm_n}_Create_UserInit(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_TAU*m*_*n*}_interrupt

INTTM*mn* 割り込みに伴う処理を実行します。

備考　この API 関数は、カウント終了割り込み（INTTM*mn*）の割り込みハンドラとして呼び出され、カウンタ値（TCR*mn*）が 0000H に達したときに発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TAUm_n}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TAUm_n}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TAUm_n}_interrupt(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

使用例

TAU チャネル 0 をディレイ・カウンタ・モードとしてカウントし、チャネル 1 を 8 ビット・ディレイ・カウンタ・モードとしてカウントする例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t ch0_run_count;
extern uint8_t ch1_run_count;

void main(void);

void main(void)
{
    EI();   // 割り込みを有効にします。
    R_Config_TAU0_0_Start();   // TAU00 動作を有効にします。
    R_Config_TAU0_0_Set_SoftwareTriggerOn();   // ソフトウェアで TS00 を 1 に設定します。
    while (ch0_run_count < 20);
    {
        // タイマ割り込み発生ごとにソフトウェアで TS00 を 1 に設定します。
        R_Config_TAU0_0_Set_SoftwareTriggerOn();
    }
    R_Config_TAU0_0_Stop();  // TAU00 動作を無効にします。

    R_Config_TAU0_1_Lower8bits_Start();  // TAU01 動作を有効にします。
    R_Config_TAU0_1_Set_SoftwareTriggerOn();  // ソフトウェアで TS01 を 1 に設定します。
    while (ch0_run_count < 20);
    {
        // タイマ割り込み発生ごとにソフトウェアで TS01 を 1 に設定します。
        R_Config_TAU0_1_Set_SoftwareTriggerOn();
    }
    R_Config_TAU0_1_Lower8bits_Stop();  // TAU01 動作を無効にします。
}
```

Config_TAU0_0_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch0_run_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_0_interrupt (void)
{
    /* Start user code for r_Config_ TAU0_0_interrupt. Do not edit comment generated here */
    ch0_run_count ++;  // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

Config_TAU0_1_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch1_run_count ;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_1_interrupt (void)
{
    /* Start user code for r_Config_TAU0_1_interrupt. Do not edit comment generated here */
    ch1_run_count ++;  // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

4.2.4 分周機能

以下に、スマート・コンフィグレータが分周機能用として出力する API 関数の一覧を示します。

表 4.7 分周機能用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TAU*m*_*n*}_Create	タイマ・アレイ・ユニット	TAU*m* チャネル・モジュールを分周器機能モードで制御する前に必要な初期化処理を実行します。
R_{Config_TAU*m*_*n*}_Start		TAU*m* のチャネル *n* のカウンタを起動します。
R_{Config_TAU*m*_*n*}_Stop		TAU*m* のチャネル *n* のカウンタを停止します。
R_{Config_TAU*m*_*n*}_Create_UserInit		TAU*m* のチャネル *n* に関するユーザ独自の初期化処理を実行します。
r_{Config_TAU*m*_*n*}_interrupt		INTTM*mn* 割り込みに伴う処理を実行します。

R_{Config_TAU*m*_*n*}_Create

TAU*m* チャネル・モジュールを分周器機能モードで制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_TAU*m*_Create から呼び出されます。

[指定形式]

```
void    R_{Config_TAUm_n}_Create(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Start

TAU*m* のチャネル *n* のカウンタを起動します。

[指定形式]

```
void    R_{Config_TAUm_n}_Start(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Stop

TAU*m* のチャネル *n* のカウンタを停止します。

[指定形式]

```
void    R_{Config_TAUm_n}_Stop(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Create_UserInit

TAU*m* のチャネル *n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TAU*m*_*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TAUm_n}_Create_UserInit(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_TAU*m*_*n*}_interrupt

INTTM*mn* 割り込みに伴う処理を実行します。

備考　この API 関数は、カウント終了割り込み（INTTM*mn*）の割り込みハンドラとして呼び出され、カウンタ値（TCR*mn*）が 0000H に達したときに発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TAUm_n}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TAUm_n}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TAUm_n}_interrupt(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

使用例

分周器機能モードでカウントする TAU チャネル 0 を使用する例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t ch0_run_count;

void main(void);

void main(void)
{
    EI();   // 割り込みを有効にします。

    R_Config_TAU0_0_Start();   // TAU00 動作を有効にします。
    while( ch0_run_count < 20);  // ch0_run_count が 20 以上になるまで待ち、ループを終了します。
    R_Config_TAU0_0_Stop();  // TAU00 動作を無効にします。
}
```

Config_TAU0_0_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch0_run_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_0_interrupt (void)
{
    /* Start user code for r_Config_ TAU0_0_interrupt. Do not edit comment generated here */
    ch0_run_count ++;  // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

4.2.5 外部イベント・カウンタ（タイマ・アレイ・ユニット）

以下に、スマート・コンフィグレータが外部イベント・カウンタ用として出力する API 関数の一覧を示します。

表 4.8 外部イベント・カウンタ（タイマ・アレイ・ユニット）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TAU*m*_*n*}_Create	タイマ・アレイ・ユニット	TAUm チャネル・モジュールを外部イベント・カウンタ・モードで制御する前に必要な初期化処理を実行します。
R_{Config_TAU*m*_*n*}_Start		TAUm のチャネル n のカウンタを起動します。
R_{Config_TAU*m*_*n*}_Stop		TAUm のチャネル n のカウンタを停止します。
R_{Config_TAUm_*n*}_Lower8bits_Start		TAUm のチャネル n の下位側 8 ビット・カウンタを起動します。
R_{Config_TAU*m*_*n*}_Lower8bits_Stop		TAUm のチャネル n の下位側 8 ビット・カウンタを停止します。
R_{Config_TAU*m*_*n*}_Create_UserInit		TAUm のチャネル n に関するユーザ独自の初期化処理を実行します。
r_{Config_TAU*m*_*n*}_interrupt		INTTMmn 割り込みに伴う処理を実行します。

R_{Config_TAU*m*_*n*}_Create

TAU*m* チャネル・モジュールを外部イベント・カウンタ・モードで制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_TAU*m*_Create から呼び出されます。

[指定形式]

```
void    R_{Config_TAUm_n}_Create(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Start

TAU*m* のチャネル n のカウンタを起動します。

[指定形式]

```
void    R_{Config_TAUm_n}_Start(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Stop

TAU*m* のチャネル n のカウンタを停止します。

[指定形式]

```
void    R_{Config_TAUm_n}_Stop(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Lower8bits_Start

TAU*m* のチャネル n の下位側 8 ビット・カウンタを起動します。

[指定形式]

```
void    R_{Config_TAUm_n}_Lower8bits_Start(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Lower8bits_Stop

TAU*m* のチャネル n の下位側 8 ビット・カウンタを停止します。

[指定形式]

```
void    R_{Config_TAUm_n}_Lower8bits_Stop(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Create_UserInit

TAU*m* のチャネル n に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TAU*m*_*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TAUm_n}_Create_UserInit(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_TAU*m*_*n*}_interrupt

INTTM*mn* 割り込みに伴う処理を実行します。

備考　この API 関数は、カウント終了割り込み（INTTM*mn*）の割り込みハンドラとして呼び出され、カウンタ値（TCR*mn*）が 0000H に達したときに発生します。

［指定形式］

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TAUm_n}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TAUm_n}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TAUm_n}_interrupt(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

［引数］

なし

［戻り値］

なし

使用例

TAU チャネル 0 を外部イベント・カウンタとしてカウントし、チャネル 1 を 8 ビット外部イベント・カウンタとしてカウントする例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t ch0_run_count;
extern uint8_t ch1_run_count;

void main(void);

void main(void)
{
    EI();    // 割り込みを有効にします。
    R_Config_TAU0_0_Start();   // TAU00 動作を有効にします。
    while (ch0_run_count < 20);  // ch0_run_count が 20 以上になるまで待ち、ループを終了します。
    R_Config_TAU0_0_Stop();   // TAU00 動作を無効にします。

    R_Config_TAU0_1_Lower8bits_Start();   // TAU01 動作を有効にします。
    while (ch1_run_count < 20);  // ch1_run_count が 20 以上になるまで待ち、ループを終了します。
    R_Config_TAU0_1_Lower8bits_Stop();   // TAU00 動作を無効にします。
}
```

Config_TAU0_0_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch0_run_count ;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_0_interrupt (void)
{
    /* Start user code for r_Config_ TAU0_0_interrupt. Do not edit comment generated here */
    ch0_run_count ++;  // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

Config_TAU0_1_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch1_run_count ;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_1_interrupt (void)
{
    /* Start user code for r_Config_TAU0_1_interrupt. Do not edit comment generated here */
    ch1_run_count ++;  // 割り込みハンドラが入力された回数をカウントします。
     /* End user code. Do not edit comment generated here */
}
```

4.2.6 外部イベント・カウンタ（タイマ RJ）

以下に、スマート・コンフィグレータが外部イベント・カウンタ（TRJO*n* 端子への入力）用として出力する API 関数の一覧を示します。

表 4.9 外部イベント・カウンタ（タイマ RJ）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TRJ*n*}_Create	タイマ RJ	タイマ RJ*n* モジュールを外部イベント・カウンタ・モードで制御する前に必要な初期化処理を実行します。
R_{Config_TRJ*n*}_Start		TRJ*n* のカウンタを起動します。
R_{Config_TRJ*n*}_Stop		TRJ*n* のカウンタを停止します。
R_{Config_TRJ*n*}_Create_UserInit		TRJ*n* に関するユーザ独自の初期化処理を実行します。
r_{Config_TRJ*n*}_interrupt		INTTRJ*n* 割り込みに伴う処理を実行します。

R_{Config_TRJ*n*}_Create

タイマ RJ*n* モジュールを外部イベント・カウンタ・モードで制御する前に必要な初期化処理を実行します。

備考　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_TRJn}_Create(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TRJ*n*}_Start

TRJ*n* のカウンタを起動します。

[指定形式]

```
void    R_{Config_TRJn}_Start(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TRJ*n*}_Stop

TRJ*n* のカウンタを停止します。

[指定形式]

```
void    R_{Config_TRJn}_Stop(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TRJ*n*}_Create_UserInit

TRJ*n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TRJ*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TRJn}_Create_UserInit(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_TRJ*n*}_interrupt

INTTRJ*n* 割り込みに伴う処理を実行します。

備考　この API 関数は、TRJ*n* 割り込み（INTTRJ*n*）の割り込みハンドラとして呼び出され、カウンタがアンダフローのときに発生します。

［指定形式］

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TRJn}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TRJn}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TRJn}_interrupt(void);
```

備考　*n* はチャネル番号を示します。

［引数］

なし

［戻り値］

なし

使用例

TRJ0 カウントをユーザ定義周期の外部イベント・カウンタとして使用する例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t ch0_run_count;

void main(void);

void main(void)
{
    EI();    // 割り込みを有効にします。
    R_Config_TRJ0_Start();   // TRJ0 動作を有効にします。
    while (ch0_run_count < 20);  // ch0_run_count が 20 以上になるまで待ち、ループを終了します。
    R_Config_TRJ0_Stop();   // TRJ0 動作を無効にします。
}
```

Config_TRJ0_0_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch0_run_count ;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TRJ0_interrupt (void)
{
    /* Start user code for r_Config_TRJ0_interrupt. Do not edit comment generated here */
    ch0_run_count ++;  // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

4.2.7 入力信号のハイ／ロウ・レベル幅測定（タイマ・アレイ・ユニット）

以下に、スマート・コンフィグレータが入力信号のハイ／ロウ・レベル幅測定用として出力する API 関数の一覧を示します。

表 4.10 入力信号のハイ／ロウ・レベル幅測定（タイマ・アレイ・ユニット）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TAU*m*_*n*}_Create	タイマ・アレイ・ユニット	入力信号のハイ／ロウ・レベル幅測定モードで TAU*m* チャネル・モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_TAU*m*_*n*}_Start		TAU*m* のチャネル *n* のカウンタを起動します。
R_{Config_TAU*m*_*n*}_Stop		TAU*m* のチャネル *n* のカウンタを停止します。
R_{Config_TAU*m*_*n*}_Get_PulseWidth		TAU*m* のチャネル *n* の入力パルス幅を測定します。
R_{Config_TAU*m*_*n*}_Create_UserInit		TAU*m* のチャネル *n* に関するユーザ独自の初期化処理を実行します。
r_{Config_TAU*m*_*n*}_interrupt		INTTM*mn* 割り込みに伴う処理を実行します。

R_{Config_TAU*m*_*n*}_Create

入力信号のハイ／ロウ・レベル幅測定モードで TAU*m* チャネル・モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_TAU*m*_Create から呼び出されます。

[指定形式]

```
void    R_{Config_TAUm_n}_Create(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Start

TAU*m* のチャネル *n* のカウンタを起動します。

[指定形式]

```
void    R_{Config_TAUm_n}_Start(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Stop

TAU*m* のチャネル *n* のカウンタを停止します。

[指定形式]

```
void    R_{Config_TAUm_n}_Stop(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Get_PulseWidth

TAU*m* のチャネル *n* の入力パルス幅を測定します。

[指定形式]

void R_{Config_TAU*m*_*n*}_Get_PulseWidth(uint32_t * const width);

備考 *m* はユニット番号、*n* はチャネル番号を示します。

[引数]

I/O	引数	説明
O	uint32_t * const width;	入力パルス幅を格納するアドレス

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Create_UserInit

TAU*m* のチャネル *n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TAU*m*_*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TAUm_n}_Create_UserInit(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_TAU*m*_*n*}_interrupt

INTTM*mn* 割り込みに伴う処理を実行します。

備考　この API 関数は、キャプチャ割り込みの割り込みハンドラ（INTTM*mn*）として呼ばれ、有効なキャプチャ・エッジが検出され、カウンタ値（TCR*mn*）がタイマ・データ・レジスタ *mn*（TDR*mn*）に転送されたときに発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TAUm_n}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TAUm_n}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TAUm_n}_interrupt(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

使用例

TI00 端子から TAU チャネル 0 入力信号のロウ・レベル幅を取得するための例です。

main.c

```
#include "r_smc_entry.h"
extern volatile uint8_t tau_interrupt_flag;
uint32_t width;    // 測定されたパルス幅を格納する変数を宣言します。

void main(void);

void main(void)
{
    EI();    // 割り込みを有効にします。
    tau_interrupt_flag = 0;    // tau_interrupt_flag の初期値を設定します。
    // TAU00 動作により、TI00 端子入力開始エッジが検出されると、カウンタは 0000H から
    カウントアップします。
    R_Config_TAU0_0_Start();

    // tau_interrupt_flag が 0 でなくなるまで待って、ループを終了します。
    これは、TI00 端子入力の有効なエッジが検出されたことを示します。
    while( tau_interrupt_flag == 0);

    R_Config_TAU0_0_Stop();   // TAU00 動作を無効にします。

    // TI00 端子から測定されたパルス幅 (ハイ・レベルまたはロウ・レベル) を取得します
    R_Config_TAU0_0_Get_PulseWidth(&width);
}
```

Config_TAU0_0_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t tau_interrupt_flag;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_0_interrupt (void)
{
    /* Start user code for r_Config_TAU0_0_interrupt. Do not edit comment generated here */
    // TI00 端子入力の有効エッジが検出され、カウント値が TDR00 に転送されます。
    tau_interrupt_flag = 1U;    /* Set the flag */
    /* End user code. Do not edit comment generated here */
}
```

4.2.8 入力信号のハイ／ロウ・レベル幅測定（タイマ RJ）

以下に、スマート・コンフィグレータが入力信号のハイ／ロウ・レベル幅（TRJO*n* 端子への入力）測定用として出力する API 関数の一覧を示します。

表 4.11 入力信号のハイ／ロウ・レベル幅測定（タイマ RJ）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TRJ*n*}_Create	タイマ RJ	入力信号のハイ／ロウ・レベル幅測定モードで TRJ*n* モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_TRJ*n*}_Start		TRJ*n* のカウンタを起動します。
R_{Config_TRJ*n*}_Stop		TRJ*n* のカウンタを停止します。
R_{Config_TRJ*n*}_Get_PulseWidth		TRJ*n* の入力パルス幅を測定します。
R_{Config_TRJ*n*}_Create_UserInit		TRJ*n* に関するユーザ独自の初期化処理を実行します。
r_{Config_TRJ*n*}_interrupt		INTTRJ*n* 割り込みに伴う処理を実行します。

R_{Config_TRJ*n*}_Create

入力信号（TRJIO*n* 端子の入力）のハイ／ロウ・レベル幅測定モードで TRJ*n* モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_TRJn}_Create(void);
```

備考　*n* = 0

[引数]

なし

[戻り値]

なし

R_{Config_TRJ*n*}_Start

TRJ*n* のカウンタを起動します。

[指定形式]

```
void    R_{Config_TRJn}_Start(void);
```

備考　　*n* = 0

[引数]

なし

[戻り値]

なし

R_{Config_TRJ*n*}_Stop

TRJ*n* のカウンタを停止します。

[指定形式]

```
void    R_{Config_TRJn}_Stop(void);
```

備考　　*n* = 0

[引数]

なし

[戻り値]

なし

R_{Config_TRJ*n*}_Get_PulseWidth

TRJ*n* の入力パルス幅を測定します。

[指定形式]

void R_{Config_TRJ*n*}_Get_PulseWidth(uint32_t * const width);

備考 *n* = 0

[引数]

I/O	引数	説明
O	uint32_t * const width;	入力パルス幅を格納するアドレス

[戻り値]

なし

R_{Config_TRJ*n*}_Create_UserInit

TRJ*n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TRJ*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TRJn}_Create_UserInit(void);
```

備考　*n* = 0

[引数]

なし

[戻り値]

なし

r_{Config_TRJ*n*}_interrupt

INTTRJ*n* 割り込みに伴う処理を実行します。

備考　この API 関数は、パルス幅測定モードで外部入力（TRJIO*n*）のアクティブ幅の測定が完了したときに発生する TRJ*n* カウンタアンダフロー割り込み（INTTRJ*n*）の割り込みハンドラとして呼び出されます。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TRJn}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TRJn}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TRJn}_interrupt(void);
```

備考　*n* = 0

[引数]

なし

[戻り値]

なし

使用例

TRJIO0 端子から TRJ0 入力パルス幅を取得するための例です。

main.c

```
#include "r_smc_entry.h"
extern volatile uint8_t interrupt_flag;
uint32_t width;    // パルス幅を格納する変数です。

void main(void);

void main(void)
{
    EI();    // 割り込みを有効にします。
    interrupt_flag = 0;    // interrupt_flag の初期値を設定します。

    // TRJIO0 端子の開始エッジが検出されると、カウンタ TRJ0 は初期値からアンダフローします。
    R_Config_TRJ0_Start();

    // interrupt_flag が 0 でなくなるまで待って、ループを終了します。
    TRJIO0 端子入力の有効なエッジが検出されたことを示します。
    while(interrupt_flag == 0);

    R_Config_TRJ0_Stop();    // TRJ0 動作を無効にします。

    // TRJIO0 端子のパルス幅は、「width」から読み取ることができます。
    R_Config_TRJ0_Get_PulseWidth(&width);
}
```

Config_TRJ0_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t interrupt_flag;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TRJ0_interrupt (void)
{
    ...
    /* Start user code for r_Config_TRJ0_interrupt. Do not edit comment generated here */
    // TRJIO0 端子入力の有効エッジが検出され、カウンタ TRJ0 が停止します。
    interrupt_flag = 1U;    /* Set the flag */
    /* End user code. Do not edit comment generated here */
}
```

4.2.9 PWM 出力（タイマ・アレイ・ユニット）

以下に、スマート・コンフィグレータが PWM 出力機能用として出力する API 関数の一覧を示します。

表 4.12 PWM 出力（タイマ・アレイ・ユニット）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TAU*m*_*n*}_Create	タイマ・アレイ・ユニット	TAU*m* チャネル・モジュールを PWM 出力モードで制御する前に必要な初期化処理を実行します。
R_{Config_TAU*m*_*n*}_Start		TAU*m* のチャネル n のカウンタを起動します。
R_{Config_TAU*m*_*n*}_Stop		TAU*m* のチャネル n のカウンタを停止します。
R_{Config_TAU*m*_*n*}_Create_UserInit		TAU*m* のチャネル n に関するユーザ独自の初期化処理を実行します。
r_{Config_TAU*m*_*n*}_channel*n*_interrupt		INTTM*mn* 割り込みに伴う処理を実行します。
r_{Config_TAU*m*_*n*}_channel*p*_interrupt		INTTM*mp* 割り込みに伴う処理を実行します。

R_{Config_TAU*m*_*n*}_Create

TAU*m* チャネル *n*・モジュールを PWM 出力モードで制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_TAU*m*_Create から呼び出されます。

[指定形式]

```
void    R_{Config_TAUm_n}_Create(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Start

TAU*m* のチャネル *n* のカウンタを起動します。

[指定形式]

```
void    R_{Config_TAUm_n}_Start(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Stop

TAU*m* のチャネル *n* のカウンタを停止します。

[指定形式]

```
void    R_{Config_TAUm_n}_Stop(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Create_UserInit

TAU*m* のチャネル *n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TAU*m*_*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TAUm_n}_Create_UserInit(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_TAU*m*_*n*}_channel*n*_interrupt

INTTM*mn* 割り込みに伴う処理を実行します。

備考　この API 関数は、カウント終了割り込み（INTTM*mn*）の割り込みハンドラとして呼び出され、カウンタ値（TCR*mn*）が 0000H に達したときに発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TAUm_n}_channeln_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TAUm_n}_channeln_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TAUm_n}_channeln_interrupt(void);
```

備考　*m* はユニット番号、*n* はマスタ・チャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_TAU*m*_*n*}_channel*p*_interrupt

INTTM*mp* 割り込みに伴う処理を実行します。

備考　この API 関数は、カウント終了割り込み（INTTM*mp*）の割り込みハンドラとして呼び出され、カウンタ値（TCR*mp*）が 0000H に達したときに発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TAUm_n}_channelp_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TAUm_n}_channelp_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TAUm_n}_channelp_interrupt(void);
```

備考 1.　*m* はユニット番号、*n* はマスタ・チャネル番号、*p* はスレーブ・チャネル番号を示します。

備考 2.　$n < p \leqq 7$

[引数]

なし

[戻り値]

なし

使用例

TAU チャネル 0/1 を開始して PWM パルスを出力する例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t tau_pwm_count
void main(void);

void main(void)
{
    EI();   // 割り込みを有効にします。
    tau_pwm_count = 0;
    R_Config_TAU0_0_Start();   // TAU0 動作を有効にします。
    while( tau_pwm_count < 10);   // tau_pwm_count が 10 以上になるまで待ち、ループを終了します。
    R_Config_TAU0_0_Stop();   // TAU0 動作を無効にします。
}
```

Config_TAU0_0_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t tau_pwm_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_1_channel1_interrupt (void)
{
    /* Start user code for r_Config_TAU0_1_channel1_interrupt. Do not edit comment generated here */
    tau_pwm_count ++;   // マスタ・チャネル割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

4.2.10 PWM 出力（PWM モード（リモコン・キャリア波形）を使用したタイマ・アレイ・ユニット）

以下に、スマート・コンフィグレータが PWM モード（リモコン・キャリア波形）機能用として出力する API 関数の一覧を示します。

表 4.13 PWM 出力（PWM モード（リモコン・キャリア波形）を使用したタイマ・アレイ・ユニット）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TAU0_*m*_TAU0_*n*}_Create	タイマ・アレイ・ユニット	TAU0 チャネル *m* およびチャネル *n*・モジュールを PWM モード（リモコン・キャリア波形）で制御する前に必要な初期化処理を実行します。
R_{Config_TAU*0*_*m*_TAU0_*n*}_Start		TAU0 チャネル *m* およびチャネル *n* のカウンタを起動します。
R_{Config_TAU*0*_*m*_TAU0_*n*}_Stop		TAU0 チャネル *m* およびチャネル *n* のカウンタを停止します。
R_{Config_TAU*0*_*m*_TAU0_*n*}_Create_UserInit		TAU0 チャネル *m* およびチャネル *n* に関するユーザ独自の初期化処理を実行します。
r_{Config_TAU*0*_*m*_TAU0_*n*}_channel*m*_interrupt		INTTM0*m* 割り込みに伴う処理を実行します。
r_{Config_TAU*0*_*m*_TAU0_*n*}_channel*p*_interrupt		INTTM0*p* 割り込みに伴う処理を実行します。
r_{Config_TAU*0*_*m*_TAU0_*n*}_channel*n*_interrupt		INTTM0*n* 割り込みに伴う処理を実行します。
r_{Config_TAU*0*_*m*_TAU0_*n*}_channel*q*_interrupt		INTTM0*q* 割り込みに伴う処理を実行します。

R_{Config_TAU0_*m*_TAU0_*n*}_Create

TAU0 チャネル *m* およびチャネル *n*・モジュールを PWM モード（リモコン・キャリア波形）で制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_TAU*m*_Create から呼び出されます。

[指定形式]

```
void    R_{Config_TAU0_m_TAU0_n}_Create(void);
```

備考　*m* はマスク波形を出力するマスタ・チャンネルの番号、*n* はキャリア波形を出力するマスタ・チャンネルの番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU0_*m*_TAU0_*n*}_Start

TAU0 チャネル *m* およびチャネル *n* のカウンタを起動します。

[指定形式]

```
void    R_{Config_TAU0_m_TAU0_n}_Start(void);
```

備考　　*m* はマスク波形を出力するマスタ・チャンネルの番号、*n* はキャリア波形を出力するマスタ・チャンネルの番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU0_*m*_TAU0_*n*}_Stop

TAU0 チャネル *m* およびチャネル *n* のカウンタを停止します。

[指定形式]

```
void    R_{Config_TAU0_m_TAU0_n}_Stop(void);
```

備考　　*m* はマスク波形を出力するマスタ・チャンネルの番号、*n* はキャリア波形を出力するマスタ・チャンネルの番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU0_*m*_TAU0_*n*}_Create_UserInit

TAU0 チャネル *m* およびチャネル *n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TAU0_*m*_TAU0_*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TAU0_m_TAU0_n}_Create_UserInit(void);
```

備考　*m* はマスク波形を出力するマスタ・チャンネルの番号、*n* はキャリア波形を出力するマスタ・チャンネルの番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_TAU0_*m*_TAU0_*n*}_channel*m*_interrupt

INTTM0*m* 割り込みに伴う処理を実行します。

備考　この API 関数は、カウント終了割り込み（INTTM0*m*）の割り込みハンドラとして呼び出され、カウンタ値（TCR0*m*）が 0000H に達したときに発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TAU0_m_TAU0_n }_channelm_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TAU0_m_TAU0_n}_channelm_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TAU0_m_TAU0_n}_channelm_interrupt(void);
```

備考　*m* はマスク波形を出力するマスタ・チャンネルの番号、*n* はキャリア波形を出力するマスタ・チャンネルの番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_TAU0_*m*_TAU0_*n*}_channel*p*_interrupt

INTTM0*p* 割り込みに伴う処理を実行します。

備考　この API 関数は、カウント終了割り込み（INTTM0*p*）の割り込みハンドラとして呼び出され、カウンタ値（TCR0*p*）が 0000H に達したときに発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TAU0_m_TAU0_n}_channelp_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TAU0_m_TAU0_n}_channelp_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TAU0_m_TAU0_n}_channelp_interrupt(void);
```

備考　*m* はマスク波形を出力するマスタ・チャンネルの番号、*n* はキャリア波形を出力するマスタ・チャンネルの番号、*p* はマスク波形を出力するスレーブ・チャンネルの番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_TAU0_*m*_TAU0_*n*}_channel*n*_interrupt

INTTM0*n* 割り込みに伴う処理を実行します。

備考　この API 関数は、カウント終了割り込み（INTTM0*n*）の割り込みハンドラとして呼び出され、カウンタ値（TCR0*n*）が 0000H に達したときに発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TAU0_m_TAU0_n}_channeln_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TAU0_m_TAU0_n}_channeln_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TAU0_m_TAU0_n}_channeln_interrupt(void);
```

備考　*m* はマスク波形を出力するマスタ・チャンネルの番号、*n* はキャリア波形を出力するマスタ・チャンネルの番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_TAU0_*m*_TAU0_*n*}_channel*q*_interrupt

INTTM0*q* 割り込みに伴う処理を実行します。

備考　この API 関数は、カウント終了割り込み（INTTM0*q*）の割り込みハンドラとして呼び出され、カウンタ値（TCR0*q*）が 0000H に達したときに発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TAU0_m_TAU0_n}_channelq_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TAU0_m_TAU0_n}_channelq_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TAU0_m_TAU0_n}_channelq_interrupt(void);
```

備考　*m* はマスク波形を出力するマスタ・チャンネルの番号、*n* はキャリア波形を出力するマスタ・チャンネルの番号、*q* はマスク波形を出力するスレーブ・チャンネルの番号を示します。

[引数]

なし

[戻り値]

なし

使用例

TAU0 channel2 と TAU0 channel4 を起動し、キャリア波計としてリモコン出力する例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t tau_remote_count;

void main(void);

void main(void)
{
    EI();   // 割り込みを有効にします。
    tau_remote_count = 0;
    R_Config_TAU0_2_TAU0_4_Start();  // リモコン出力機能を有効にします。
    while (tau_remote_count < 10);  // tau_remote_count が 10 以上になるまで待ち、ループを終了します。
    R_Config_TAU0_2_TAU0_4_Stop();  // リモコン出力機能を無効にします。
}
```

Config_TAU0_2_TAU0_4_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t tau_remote_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_2_TAU0_4_channel3_interrupt (void)
{
    /* Start user code for r_Config_ TAU0_2_TAU0_4_channel3_interrupt. Do not edit comment generated here */
    tau_remote_count ++;   // マスク波形割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

4.2.11 PWM 出力（PWM モード／拡張 PWM モードを使用したタイマ RD*n*）

以下に、スマート・コンフィグレータが PWM 波形を出力するために出力する API 関数の一覧を示します。

表 4.14 PWM 出力（PWM モード／拡張 PWM モードを使用したタイマ RD*n*）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TRD*n*}_Create	タイマ RD	TRD*n* モジュールを PWM モード／拡張 PWM モードで制御する前に必要な初期化処理を実行します。
R_{Config_TRD*n*}_Start		TRD*n* のカウンタを起動します。
R_{Config_TRD*n*}_Stop		TRD*n* のカウンタを停止します。
R_{Config_TRD*n*}_Set_TRD*n*_ReloadTriger		拡張 PWM モードで TRD*n* バッファレジスタのリロードトリガを生成します。
R_{Config_TRD*n*}_Create_UserInit		TRD*n* に関するユーザ独自の初期化処理を実行します。
r_{Config_TRD*n*}_trd*n*_interrupt		INTRD*n* 割り込みに伴う処理を実行します。

R_{Config_TRD*n*}_Create

TRD*n* モジュールを PWM モード／拡張 PWM モードで制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_TRD_Create から呼び出されます。

[指定形式]

```
void    R_{Config_TRDn}_Create(void);
```

備考　*n* = 0, 1

[引数]

なし

[戻り値]

なし

R_{Config_TRD*n*}_Start

TRD*n* のカウンタを起動します。

[指定形式]

```
void    R_{Config_TRDn}_Start(void);
```

備考　　*n* = 0, 1

[引数]

なし

[戻り値]

なし

R_{Config_TRD*n*}_Stop

TRD*n* のカウンタを停止します。

[指定形式]

```
void    R_{Config_TRDn}_Stop(void);
```

備考　　*n* = 0, 1

[引数]

なし

[戻り値]

なし

R_{Config_TRD*n*}_Set_TRD*n*_ReloadTrigger

拡張 PWM モードで TRD*n* バッファレジスタのリロードトリガを生成します。

[指定形式]

```
MD_STATUS R_{Config_TRDn}_Set_TRDn_ReloadTrigger (st_extpwm_buffer_registers_t * buffer);
```

備考 *n* = 0, 1

[引数]

I/O	引数	説明
I	st_extpwm_buffer_registers_t * buffer;	バッファレジスタ値

備考 構造体 st_extpwm_buffer_registers_t を以下に示します。

```
typedef struct {
  uint16_t trdgrcn;
  uint16_t trdgrdn;
  uint16_t trdcmpdn;
} st_extpwm_buffer_registers_t;
```

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_ERROR	リロードトリガステータス待ち

R_{Config_TRD*n*}_Create_UserInit

TRD*n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TRD*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TRDn}_Create_UserInit(void);
```

備考　*n* = 0, 1

[引数]

なし

[戻り値]

なし

r_{Config_TRD*n*}_trd*n*_interrupt

INTRD*n* 割り込みに伴う処理を実行します。

備考　この API 関数は、TRD*n* レジスタの内容が TRDGR*hn*（h = A、B、C、または D）レジスタの内容と一致するか、TRD*n* レジスタがオーバーフローの場合に発生するカウント・コンペアー致割り込み（INTRD*n*）の割り込みハンドラとして呼び出されます。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TRDn}_trdn_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TRDn}_trdn_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TRDn}_trdn_interrupt(void);
```

備考　*n* = 0, 1

[引数]

なし

[戻り値]

なし

使用例

TRD0 を起動して 3 つの PWM 波形パルスを出力し、TRDGRA0、TRDGRB0、TRDCMPB0 レジスタ値を同時に更新する例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t trd_pwm_count;

void main(void);

st_extpwm_buffer_registers_t buffer;    // タイマ RD0 のバッファレジスタ値に書き込みます。
buffer.trdgrcn = 0x1234;
buffer.trdgrdn = 0x5678;
buffer.trdcmpdn = 0x9ABC;

static void delay_ms(uint32_t time_ms)
{
    uint32_t i = 0;
    while(time_ms--) {
        for(i = 0; i < 156; i++) {
            NOP();
        }
    }
}

void main(void)
{
    EI();   // 割り込みを有効にします。
    R_Config_TRD0_Start();   // TRD0 動作を有効にします。
    while (trd_pwm_count < 10);  // tau_pwm_count が 10 以上になるまで待ち、ループを終了します。
    delay_ms(200);
    MD_STATUS result = R_Config_TRD0_Set_TRD0_ReloadTrigger(&buffer);    // バッファ・レジスタ値をトリガしてリロードします。
    while (status != MD_OK);
    delay_ms(2000);
    R_Config_TRD0_Stop();   // TRD0 動作を無効にします。
}
```

Config_TRD0_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t trd_pwm_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TRD0_trd0_interrupt (void)
{
    /* Start user code for r_Config_TRD0_trd0_interrupt. Do not edit comment generated here */
    trd_pwm_count ++;   // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

4.2.12 PWM 出力（PWM モード／PWM3 モード／拡張 PWM モード／タイマ KB3 PWM 出力ゲートモードを使用したタイマ RD0 と RD1）

以下に、スマート・コンフィグレータが PWM 出力機能の PWM3 モード／タイマ KB3 PWM 出力ゲートモードでの 2 波形、タイマ KB3 PWM 出力ゲートモードでの 4 波形、または PWM モード／拡張 PWM モード／タイマ KB3 PWM 出力ゲートモードでの 6 波形として出力する API 関数の一覧を示します。

表 4.15 PWM 出力（PWM モード／PWM3 モード／拡張 PWM モードを使用したタイマ RD）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TRD0_TRD1}_Create	タイマ RD	TRD0 モジュールを PWM3 モードで、または TRD*n* モジュールを PWM モード／拡張 PWM モード／タイマ KB3 PWM 出力ゲートモードで制御する前に必要な初期化処理を実行します。
R_{Config_TRD0_TRD1}_Start		PWM3 モードで TRD0 カウンタを起動するか、PWM モード／拡張 PWM モード／タイマ KB3 PWM 出力ゲートモードで TRD*n* カウンタを起動します。
R_{Config_TRD0_TRD1}_Stop		PWM3 モードで TRD0 カウンタを停止するか、PWM モード／拡張 PWM モード／タイマ KB3 PWM 出力ゲートモードで TRD*n* カウンタを停止します。
R_{Config_TRD0_TRD1}_Set_TRD*n*_ReloadTriger		拡張 PWM モードで TRD*n* バッファレジスタのリロードトリガを生成します。
R_{Config_TRD0_TRD1}_Set_TRD0_ReloadTriger		タイマ KB3 PWM 出力ゲートモードで TRD0 バッファレジスタのリロードトリガを生成します。
R_{Config_TRD0_TRD1}_Set_TRD1_ReloadTriger		タイマ KB3 PWM 出力ゲートモードで TRD1 バッファレジスタのリロードトリガを生成します。
R_{Config_TRD0_TRD1}_Create_UserInit		TRD0_TRD1 に関するユーザ独自の初期化処理を実行します。
r_{Config_TRD0_TRD1}_trd*n*_interrupt		INTRD*n* 割り込みに伴う処理を実行します。

R_{Config_TRD0_TRD1}_Create

TRD0 モジュールを PWM3 モードで、または TRD*n* モジュールを PWM モード／拡張 PWM モード／タイマ KB3 PWM 出力ゲートモードで制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_TRD_Create から呼び出されます。

[指定形式]

```
void    R_{Config_TRD0_TRD1}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TRD0_TRD1}_Start

PWM3 モードで TRD0 カウンタを起動するか、PWM モード／拡張 PWM モード／タイマ KB3 PWM 出力ゲートモードで TRD*n* カウンタを起動します。

[指定形式]

```
void    R_{Config_TRD0_TRD1}_Start(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TRD0_TRD1}_Stop

PWM3 モードで TRD0 カウンタを停止するか、PWM モード／拡張 PWM モード／タイマ KB3 PWM 出力ゲートモードで TRD*n* カウンタを停止します。

［指定形式］

```
void    R_{Config_TRD0_TRD1}_Stop(void);
```

［引数］

なし

［戻り値］

なし

R_{Config_TRD0_TRD1}_Set_TRD*n*_ReloadTrigger

拡張 PWM モードで TRD*n* バッファレジスタのリロードトリガを生成します。

[指定形式]

```
MD_STATUS  R_{Config_TRD0_TRD1}_Set_TRDn_ReloadTrigger  (st_extpwm_buffer_registers_t
* buffer);
```

備考　　*n* = 0, 1

[引数]

I/O	引数	説明
I	st_extpwm_buffer_registers_t * buffer;	バッファレジスタ値

備考　　構造体 st_extpwm_buffer_registers_t を以下に示します。

```
typedef struct {
   uint16_t trdgrcn;
   uint16_t trdgrdn;
   uint16_t trdcmpdn;
} st_extpwm_buffer_registers_t;
```

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_ERROR	リロードトリガステータス待ち

R_{Config_TRD0_TRD1}_Set_TRD0_ReloadTrigger

タイマ KB3 PWM 出力ゲートモードで TRD0 バッファレジスタのリロードトリガを生成します。

[指定形式]

```
MD_STATUSR_{Config_TRD0_TRD1}_Set_TRD0_ReloadTrigger
(st_ kb3pwm_ch0_buffer_registers_t * buffer);
```

備考　　*n* = 0, 1

[引数]

I/O	引数	説明
I	st_kb3pwm_ch0_buffer_registers_t * buffer;	バッファレジスタ値

備考　　構造体 st_kb3pwm_ch0_buffer_registers_t を以下に示します。

```
typedef struct {
   uint16_t trdgra0;
   uint16_t trdgrb0;
   uint16_t trdcmpb0;
} st_kb3pwm_ch0_buffer_registers_t;
```

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_ERROR	リロードトリガステータス待ち

R_{Config_TRD0_TRD1}_Set_TRD1_ReloadTrigger

タイマ KB3 PWM 出力ゲートモードで TRD1 バッファレジスタのリロードトリガを生成します。

[指定形式]

```
MD_STATUSR_{Config_TRD0_TRD1}_Set_TRD1_ReloadTrigger
(st_ kb3pwm_ch1_buffer_registers_t * buffer);
```

備考　　*n* = 0, 1

[引数]

I/O	引数	説明
I	st_kb3pwm_ch1_buffer_registers_t * buffer;	バッファレジスタ値

備考　　構造体 st_kb3pwm_ch1_buffer_registers_t を以下に示します。

```
typedef struct {
   uint16_t trdgra1;
   uint16_t trdgrb1;
   uint16_t trdcmpa1;
   uint16_t trdcmpb1;
   uint16_t trdcmpe1;
} st_kb3pwm_ch1_buffer_registers_t;
```

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_ERROR	リロードトリガステータス待ち

R_{Config_TRD0_TRD1}_Create_UserInit

TRD0_TRD1 に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TRD0_TRD1}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TRD0_TRD1}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_TRD0_TRD1}_trd*n*_interrupt

INTTRD*n* 割り込みに伴う処理を実行します。

備考　この API 関数は、TRD*n* レジスタの内容が TRDGR*hn*（h = A、B、C、または D）レジスタの内容と一致するか、PWM モード／PWM3 モード／拡張 PWM モード／タイマ KB3 PWM 出力ゲートモードで TRD*n* レジスタがオーバーフローの場合に発生するカウント・コンペア一致割り込み（INTRD*n*）の割り込みハンドラとして呼び出されます。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TRD0_TRD1}_trdn_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TRD0_TRD1}_trdn_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TRD0_TRD1}_trdn_interrupt(void);
```

備考　*n* = 0, 1

[引数]

なし

[戻り値]

なし

使用例

TRD0_TRD1 を起動して 6 つの PWM 波形パルスを出力する例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t trd_kb3_pwm_count;

void main(void);

void main(void)
{
    EI();   // 割り込みを有効にします。
    R_Config_TRD0_TRD1_Start();   // TRD0 および TRD1 動作を有効にします。
    while (trd_kb3_pwm_count < 20);   // trd_kb3_pwm_count が 20 以上になるまで待ち、ループを終了します。
    R_Config_TRD0_TRD1_Stop();   // TRD0 および TRD1 動作を無効にします。
}
```

Config_TRD0_TRD1_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t trd_kb3_pwm_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TRD0_TRD1_trd0_interrupt (void)
{
    /* Start user code for r_Config_TRD0_trd0_interrupt. Do not edit comment generated here */
    trd_kb3_pwm_count ++;   // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}

static void __near r_Config_TRD0_TRD1_trd1_interrupt (void)
{
    /* Start user code for r_Config_TRD0_trd1_interrupt. Do not edit comment generated here */
    trd_kb3_pwm_count ++;   // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

4.2.13 PWM 出力（PWM モード／PWM2 モードを使用したタイマ RG）

以下に、スマート・コンフィグレータが PWM 出力機能用として出力する API 関数の一覧を示します。

表 4.16 PWM 出力（PWM モード／PWM2 モードを使用したタイマ RG）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TRG}_Create	タイマ RG	TRG モジュールを PWM モード／PWM2 モードで制御する前に必要な初期化処理を実行します。
R_{Config_TRG}_Start		TRG カウンタを起動します。
R_{Config_TRG}_Stop		TRG カウンタを停止します。
R_{Config_TRG}_Create_UserInit		TRG の初期化処理を実行します。
r_{Config_TRG}_interrupt		タイマ RG がコンペアー致割り込み(INTTRG)に伴う処理を実行します。

R_{Config_TRG}_Create

TRG を PWM モード／PWM2 モードで制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、main()関数が実行される前に R_Systeminit から呼び出されます。

[指定形式]

```
void    R_TRG_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TRG}_Start

TRG のカウンタを起動します。

[指定形式]

```
void    R_{Config_TRG}_Start(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TRG}_Stop

TRG のカウンタを停止します。

[指定形式]

```
void    R_{Config_TRG}_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TRG}_Create_UserInit

TRG に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TRG}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TRG}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_TRG}_interrupt

タイマ RG がカウント・コンペアー致割り込み(INTTRG)に伴う処理を実行します。

備考　この API 関数は、カウント・コンペアー致割り込みの割り込みハンドラ（INTTRG）として呼ばれ、TRG レジスタの内容が TRGGR*h* (h = A、B、C、または D) レジスタの内容と一致するか、TRG レジスタのオーバーフローに発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TRG}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TRG}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TRG}_interrupt(void);
```

[引数]

なし

[戻り値]

なし

使用例

TRG を起動して PWM 波形パルスを出力する例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t trg_pwm_count;

void main(void);

void main(void)
{
    EI();   // 割り込みを有効にします。
    R_Config_TRG_Start();   // TRG 動作を有効にします。
    while (trg_pwm_count < 10);  // trd_pwm_count が 10 以上になるまで待ち、ループを終了します。
    R_Config_TRG_Stop();   // TRG 動作を無効にします。
}
```

Config_TRG_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t trg_pwm_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TRG_interrupt (void)
{
    /* Start user code for r_Config_TRG_interrupt. Do not edit comment generated here */
    trg_pwm_count ++;   // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

4.2.14 PWM 出力（単体動作モード（TKBCR*n*0 レジスタによる周期制御）／単体動作モード（外部トリガ入力による周期制御）／インターリーブ PFC 出力モードを使用したタイマ KB）

以下に、スマート・コンフィグレータが PWM 出力機能用として出力する API 関数の一覧を示します。

表 4.17 PWM 出力（単体動作モード（TKBCR*n*0 レジスタによる周期制御）／単体動作モード（外部トリガ入力による周期制御）／インターリーブ PFC 出力モードを使用したタイマ KB）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TKB*n*}_Create	タイマ KB	TKB*n* モジュールを単体動作モード（TKBCRn0 レジスタによる周期制御）／単体動作モード（外部トリガ入力による周期制御）／インターリーブ PFC 出力モードで制御する前に必要な初期化処理を実行します。
R_{Config_TKB*n*}_Start		TKB*n* カウンタを起動します。
R_{Config_TKBn}_Stop		TKB*n* カウンタを停止します。
R_{Config_TKBn}_Set_BatchOverwrite RequesetOn		TKB*n* の一斉書き込みを動作します。
R_{Config_TKBn}_TKBOn0_Forced_Output_Stop_Function1_Start		TKBn TKBO*n*0 の強制出力停止機能 1 を開始します。
R_{Config_TKBn}_TKBOn0_Forced_Output_Stop_Function1_Stop		TKBn TKBO*n*0 の強制出力停止機能 1 を停止します。
R_{Config_TKBn}_TKBOn1_Forced_Output_Stop_Function1_Start		TKBn TKBO*n*1 の強制出力停止機能 1 を開始します。
R_{Config_TKBn}_TKBOn1_Forced_Output_Stop_Function1_Stop		TKBn TKBO*n*1 の強制出力停止機能 1 を停止します。
R_{Config_TKBn}_TKBOn0_SmoothStartFunction_Start		TKBn TKBO*n*0 のソフト・スタート機能を開始します。
R_{Config_TKBn}_TKBOn0_SmoothStartFunction_Stop		TKBn TKBO*n*0 のソフト・スタート機能を停止します。
R_{Config_TKBn}_TKBOn1_SmoothStartFunction_Start		TKB*n* TKBO*n*1 のソフト・スタート機能を開始します。
R_{Config_TKBn}_TKBOn1_SmoothStartFunction_Stop		TKBn TKBO*n*1 のソフト・スタート機能を停止します。
R_{Config_TKBn}_Create_UserInit		TKB*n* に関するユーザ独自の初期化処理を実行します。
r_{Config_TKBn}_terminated0_interrupt		TKB*n* TKBO*n*0 の強制出力停止解除割り込み（INTTMKBSTP*n*0）に伴う処理を実行します。
r_{Config_TKBn}_terminated1_interrupt		TKB*n* TKBO*n*1 の強制出力停止解除割り込み（INTTMKBSTP*n*1）に伴う処理を実行します。
r_{Config_TKBn}_activated0_interrupt		TKB*n* TKBO*n*0 の強制出力停止発動割り込み（INTTMKBSTR*n*0）に伴う処理を実行します。
r_{Config_TKBn}_activated1_interrupt		TKB*n* TKBO*n*1 の強制出力停止発動割り込み（INTTMKBSTR*n*1）に伴う処理を実行します。
r_{Config_TKBn}_end_count_interrupt		TKB*n* カウント・コンペアー致割り込み（INTTMKB*n*）の割り込みに伴う処理を実行します。

R_{Config_TKB*n*}_Create

TKB*n* モジュールを単体動作モード（TKBCRn0 レジスタによる周期制御）／単体動作モード（外部トリガ入力による周期制御）／インターリーブ PFC 出力モードで制御する前に必要な初期化処理を実行します。

備考　この API 関数は、main()関数が実行される前に R_TKB_Create から呼び出されます。

[指定形式]

```
void    R_TRG_Create(void);
```

備考　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB*n*}_Start

TKB*n* のカウンタを起動します。

[指定形式]

```
void    R_{Config_TKBn}_Start(void);
```

備考　　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB*n*}_Stop

TKB*n* のカウンタを停止します。

[指定形式]

```
void    R_{Config_TKBn}_Stop(void);
```

備考　　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB*n*}_Set_BatchOverwriteRequestOn

TKB*n* の一斉書き込み要求関数を設定します。

[指定形式]

```
void    R_{Config_TKBn}_Set_BatchOverwriteRequestOn (void);
```

備考　　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB*n*}_TKBO*n*0_Forced_Output_Stop_Function1_Start

TKB*n* TKBO*n*0 の強制出力停止機能 1 を開始します。

[指定形式]

```
void    R_{Config_TKBn}_TKBOn0_Forced_Output_Stop_Function1_Start (void);
```

備考　　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB*n*}_TKBO*n*0_Forced_Output_Stop_Function1_Stop

TKB*n* TKBO*n*0 の強制出力停止機能 1 を停止します。

[指定形式]

```
void    R_{Config_TKBn}_TKBOn0_Forced_Output_Stop_Function1_Stop (void);
```

備考　　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB*n*}_TKBO*n*1_Forced_Output_Stop_Function1_Start

TKB*n* TKBO*n*1 の強制出力停止機能 1 を開始します。

[指定形式]

```
void    R_{Config_TKBn}_TKBOn1_Forced_Output_Stop_Function1_Start (void);
```

備考　　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB*n*}_TKBO*n*1_Forced_Output_Stop_Function1_Stop

TKB*n* TKBO*n*1 の強制出力停止機能 1 を停止します。

[指定形式]

```
void    R_{Config_TKBn}_TKBOn1_Forced_Output_Stop_Function1_Stop (void);
```

備考　　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB*n*}_TKBO*n*0_SmoothStartFunction_Start

TKB*n* TKBO*n*0 のソフト・スタート機能を開始します。

[指定形式]

```
void    R_{Config_TKBn}_TKBOn0_SmoothStartFunction_Start (void);
```

備考　　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB*n*}_TKBO*n*0_SmoothStartFunction_Stop

TKB*n* TKBO*n*0 のソフト・スタート機能を停止します。

[指定形式]

void R_{Config_TKB*n*}_TKBO*n*0_SmoothStartFunction_Stop (void);

備考 *n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB*n*}_TKBO*n*1_SmoothStartFunction_Start

TKB*n* TKBO*n*1 のソフト・スタート機能を開始します。

[指定形式]

```
void    R_{Config_TKBn}_TKBOn1_SmoothStartFunction_Start (void);
```

備考　　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB*n*}_TKBO*n*1_SmoothStartFunction_Stop

TKB*n* TKBO*n*1 のソフト・スタート機能を停止します。

[指定形式]

```
void    R_{Config_TKBn}_TKBOn1_SmoothStartFunction_Stop (void);
```

備考　　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB*n*}_Create_UserInit

TKB*n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TKB*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TKBn}_Create_UserInit(void);
```

備考　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

r_{Config_TKB*n*}_terminated0_interrupt

TKB*n* TKBO*n*0 の強制出力停止解除割り込み（INTTMKBSTP*n*0）に伴う処理を実行します。

［指定形式］

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TKBn}_ terminated0_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TKBn}_ terminated0_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TKBn}_ terminated0_interrupt(void);
```

備考　　*n* = 0, 1, 2

［引数］

なし

［戻り値］

なし

r_{Config_TKB*n*}_terminated1_interrupt

TKB*n* TKBO*n*1 の強制出力停止解除割り込み（INTTMKBSTP*n*1）に伴う処理を実行します。

［指定形式］

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TKBn}_ terminated1_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TKBn}_ terminated1_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TKBn}_ terminated1_interrupt(void);
```

備考　　*n* = 0, 1, 2

［引数］

なし

［戻り値］

なし

r_{Config_TKB*n*}_activated0_interrupt

TKB*n* TKBO*n*0 の強制出力停止発動割り込み（INTTMKBSTR*n*0）に伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TKBn}_activated0_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TKBn}_activated0_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TKBn}_ activated0_interrupt(void);
```

備考　　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

r_{Config_TKB*n*}_activated1_interrupt

TKB*n* TKBO*n*1 の強制出力停止発動割り込み（INTTMKBSTR*n*1）に伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TKBn}_activated1_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TKBn}_activated1_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TKBn}_ activated1_interrupt(void);
```

備考　　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

r_{Config_TKB*n*}_end_count_interrupt

タイマ KB*n* カウント・コンペアー致割り込み（INTTMKBn）の割り込みに伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TRD n}_end_count _interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TRD n}_end_count_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TRD n}_end_count_interrupt(void);
```

備考　　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

使用例 1 (ソフト・スタート機能)

TKB1 を起動して PWM 波形パルスを出力し、ソフト・スタート機能を出力する例です。
まず、スマート・コンフィグレータで TKBO10 の PWM 出力ソフト・スタート機能を設定してください。

main.c

```
#include "r_smc_entry.h"
extern uint8_t tkb_pwm_count;

void main(void);

// ミリ秒レベルの遅延をシミュレーションする遅延機能。
static void delay_ms(uint32_t time_ms)
{
    uint32_t i = 0;
    while(time_ms--) {
        for(i = 0; i < 156; i++) {
            NOP();
        }
    }
}

void main(void)
{
    EI();   // 割り込みを有効にします。
    R_Config_TKB1_TKBO10_SmoothStartFunction_Stop();    // TKB1 の TKBO10 ソフト・スタート機能を停止します。
    R_Config_TKB1_Start();   // TKB1 動作を有効にします。
    while (tkb_pwm_count < 10);   // PWM 波形パルスを出力し、デューティ値が TKBSIR10 レジスタ値と同じになります。
    R_Config_TKB1_Stop();   // TKB1 動作を無効にします。
    delay_ms(500);
    R_Config_TKB1_TKBO10_SmoothStartFunction_Start();    // TKB1 の TKBO10 ソフト・スタート機能を開始します。
    R_Config_TKB1_Start();   // TKB1 動作を再開します。
    delay_ms(10000);    // PWM 波形パルスを出力し、最初はデューティ値が TKBSSR10 レジスタ値と同じになります。しばらくすると、ソフト・スタート機能が完了し、デューティ値は TKBSIR10 レジスタ値に変更されます。
    R_Config_TKB1_Stop();   // TKB1 動作を無効にします。
}
```

Config_TKB1_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t tkb_pwm_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TKB1_end_count_interrupt (void)
{
    /* Start user code for r_Config_TKB1_end_count_interrupt. Do not edit comment generated here */
    tkb_pwm_count ++;   // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

使用例 2 (コンペア・レジスター斉書き換え機能)

PWM 波形パルス出力のコンペア・レジスター斉書き換え機能による周期とデューティの変更例です。

main.c

```
#include "r_cg_macrodriver.h"
#include "Config_TKB0.h"

void main(void);

// ミリ秒レベルの遅延をシミュレーションする遅延機能。
static void delay_ms(uint32_t time_ms)
{
	uint32_t i = 0;
	while(time_ms--) {
		for(i = 0; i < 156; i++) {
			NOP();
		}
	}
}

void main(void)
{
	EI();	// 割り込みを有効にします。

	//TKB0 動作を開始します。
	R_Config_TKB0_Start();

	// 一斉書き換えトリガをオンにします。
	delay_ms(100);
	TKBCR00 = 0x0C7FU;	// PWM 周期を 100us に変更します。
	TKBCR01 = 0x0A00U;	// デューティ (TKBO00) を 80%に変更します。
	TKBCR02 = 0x0280U;	// デューティ (TKBO01) を 80%に変更します。
	TKBCR03 = 0x0C80U;	// デューティ (TKBO01) を 20%に変更します。
	R_Congif_TKB0_Set_BatchOverwriteRequestOn();	// 一斉書き換えトリガをオンにします。
	P14_bit.no0 = ~P14_bit.no0;	// 開始タイミングでの一斉書き換えトリガを確認します。

	while(1);
}
```

4.2.15 PWM 出力（同時スタート／ストップ・モード（TKBCRn0 レジスタによる周期制御）／同時スタート／ストップ・モード（外部トリガ入力による周期制御）／同時スタート／ストップ・モード（マスタによる周期制御）を使用したタイマ KB）（1 スレーブ）

以下に、スマート・コンフィグレータが PWM 出力機能用として出力する API 関数の一覧を示します。

表 4.18 PWM 出力（同時スタート／ストップ・モード（TKBCR*n*0 レジスタによる周期制御）／同時スタート／ストップ・モード（外部トリガ入力による周期制御）／同時スタート／ストップ・モード（マスタによる周期制御）を使用したタイマ KB）用 API 関数（1/2）

API 関数名	周辺機能	機能概要
R_{Config_TKB0_TKBn}_Create	タイマ KB	TKB0 と TKB*n* モジュールを同時スタート／ストップ・モード（TKBCR*n*0 レジスタによる周期制御）／同時スタート／ストップ・モード（外部トリガ入力による周期制御）／同時スタート／ストップ・モード（マスタによる周期制御）で制御する前に必要な初期化処理を実行します。
R_{Config_TKB0_TKBn}_Start		TKB0 と TKB*n* カウンタを起動します。
R_{Config_TKB0_TKBn}_Stop		TKB0 と TKB*n* カウンタを停止します。
R_{Config_TKB0_TKBn}_TKBm_Set_BatchOverwriteRequestOn		TKB*m* の一斉書き込みを動作します。
R_{Config_TKB0_TKBn}_TKBOm0_Forced_Output_Stop_Function1_Start		TKB*m* TKBO*m*0 の強制出力停止機能 1 を開始します。
R_{Config_TKB0_TKBn}_TKBOm0_Forced_Output_Stop_Function1_Stop		TKB*m* TKBO*m*0 の強制出力停止機能 1 を停止します。
R_{Config_TKB0_TKBn}_TKBOm1_Forced_Output_Stop_Function1_Start		TKB*m* TKBO*m*1 の強制出力停止機能 1 を開始します。
R_{Config_TKB0_TKBn}_TKBOm1_Forced_Output_Stop_Function1_Stop		TKB*m* TKBO*m*1 の強制出力停止機能 1 を停止します。
R_{Config_TKB0_TKBn}_TKBOm0_SmoothStartFunction_Start		TKB*m* TKBO*m*0 のソフト・スタート機能を開始します。
R_{Config_TKBn}_TKBOn0_SmoothStartFunction_Stop		TKBm TKBO*m*0 のソフト・スタート機能を停止します。
R_{Config_TKB0_TKBn}_TKBOm1_SmoothStartFunction_Start		TKB*m* TKBO*m*1 のソフト・スタート機能を開始します。
R_{Config_TKB0_TKBn}_TKBOm1_SmoothStartFunction_Stop		TKB*m* TKBO*m*1 のソフト・スタート機能を停止します。

表 4.19 PWM 出力（同時スタート／ストップ・モード（TKBCR*n*0 レジスタによる周期制御）／同時スタート／ストップ・モード（外部トリガ入力による周期制御）／同時スタート／ストップ・モード（マスタによる周期制御）を使用したタイマ KB）API 関数（2/2）

API 関数名	周辺機能	機能概要
R_{Config_TKB0_TKBn}_Create_UserInit		TKB0 と TKB*n* に関するユーザ独自の初期化処理を実行します。
r_{Config_TKB0_TKBn}_tkbm_terminated0_interrupt		TKB*m* TKBO*m*0 の強制出力停止解除割り込み（INTTMKBSTP*m*0）に伴う処理を実行します。
r_{Config_TKB0_TKBn}_tkbm_terminated1_interrupt		TKB*m* TKBO*m*1 の強制出力停止解除割り込み（INTTMKBSTP*m*1）に伴う処理を実行します。
r_{Config_TKB0_TKBn}_tkbm_activated0_interrupt		TKB*m* TKBO*m*0 の強制出力停止発動割り込み（INTTMKBSTR*m*0）に伴う処理を実行します。
r_{Config_TKB0_TKBn}_tkbm_activated1_interrupt		TKB*m* TKBO*m*1 の強制出力停止発動割り込み（INTTMKBSTR*m*1）に伴う処理を実行します。
r_{Config_TKB0_TKBn}_tkbm_end_count_interrupt		TKB*m* カウント・コンペアー致割り込み（INTTMKBm）の割り込みに伴う処理を実行します。

R_{Config_TKB0_TKB*n*}_Create

TKB0 と TKB*n* モジュールを同時スタート／ストップ・モード（TKBCRn0 レジスタによる周期制御）／同時スタート／ストップ・モード（外部トリガ入力による周期制御）／同時スタート／ストップ・モード（マスタによる周期制御）で制御する前に必要な初期化処理を実行します。

備考　この API 関数は、main()関数が実行される前に R_TKB_Create から呼び出されます。

[指定形式]

```
void    R_{Config_TKB0_TKBn}_Create(void);
```

備考　*n* = 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB*n*}_Start

TKB0 と TKB*n* のカウンタを起動します。

[指定形式]

void R_{Config_TKB0_TKB*n*}_Start(void);

備考 *n* = 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB*n*}_Stop

TKB0 と TKB*n* のカウンタを停止します。

[指定形式]

void R_{Config_TKB0_TKB*n*}_Stop(void);

備考 *n* = 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB*n*}_TKB*m*_Set_BatchOverwriteRequestOn

TKB*m* の一斉書き込みを動作します。

[指定形式]

void R_{Config_TKB0_TKB*n*}_TKB*m*_Set_BatchOverwriteRequestOn (void);

備考 *n* = 1, 2 *m* = 0, *n*

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB*n*}_TKBO*m*0_Forced_Output_Stop_Function1_Start

KB*m* TKBO*m*0 の強制出力停止機能 1 を開始します。

[指定形式]

```
void    R_{Config_TKB0_TKBn}_TKBOm0_Forced_Output_Stop_Function1_Start (void);
```

備考　　*n* = 1, 2　*m* = 0, *n*

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB*n*}_TKBO*m*0_Forced_Output_Stop_Function1_Stop

KB*m* TKBO*m*0 の強制出力停止機能 1 を停止します。

[指定形式]

```
void    R_{Config_TKB0_TKBn}_TKBOm0_Forced_Output_Stop_Function1_Stop (void);
```

備考　　*n* = 1, 2　*m* = 0, *n*

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB*n*}_TKBO*m*1_Forced_Output_Stop_Function1_Start

KB*m* TKBO*m*1 の強制出力停止機能 1 を開始します。

[指定形式]

```
void    R_{Config_TKB0_TKBn}_TKBOm1_Forced_Output_Stop_Function1_Start (void);
```

備考　　*n* = 1, 2　*m* = 0, *n*

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB*n*}_TKBO*m*1_Forced_Output_Stop_Function1_Stop

KB*m* TKBO*m*1 の強制出力停止機能 1 を停止します。

[指定形式]

```
void    R_{Config_TKB0_TKBn}_TKBOm1_Forced_Output_Stop_Function1_Stop (void);
```

備考　　*n* = 1, 2　*m* = 0, *n*

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB*n*}_TKBO*m*0_SmoothStartFunction_Start

TKB*m* TKBO*n*0 のソフト・スタート機能を開始します。

[指定形式]

void R_{Config_TKB0_TKB*n*}_TKBO*m*0_SmoothStartFunction_Start (void);

備考 *n* = 1, 2 *m* = 0, *n*

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB*n*}_TKBO*m*0_SmoothStartFunction_Stop

TKB*m* TKBO*m*0 のソフト・スタート機能を停止します。

[指定形式]

void R_{Config_TKB0_TKB*n*}_TKBO*m*0_SmoothStartFunction_Stop (void);

備考 *n* = 1, 2 *m* = 0, *n*

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB*n*}_TKBO*m*1_SmoothStartFunction_Start

TKB*m* TKBO*m*1 のソフト・スタート機能を開始します。

[指定形式]

void R_{Config_TKB0_TKB*n*}_TKBO*m*1_SmoothStartFunction_Start (void);

備考 *n* = 1, 2 *m* = 0, *n*

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB*n*}_TKBO*m*1_SmoothStartFunction_Stop

TKB*m* TKBO*m*1 のソフト・スタート機能を停止します。

[指定形式]

void R_{Config_TKB0_TKB*n*}_TKBO*m*1_SmoothStartFunction_Stop (void);

備考 *n* = 1, 2 *m* = 0, *n*

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB*n*}_Create_UserInit

TKB0 と TKB*n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TKB0_TKBn}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TKB0_TKBn}_Create_UserInit(void);
```

備考　*n* = 1, 2　*m* = 0, *n*

[引数]

なし

[戻り値]

なし

r_{Config_TKB0_TKB*n*}_tkb*m*_terminated0_interrupt

TKB*m* TKBO*m*0 の強制出力停止解除割り込み（INTTMKBSTP*m*0）に伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TKB0_TKBn}_tkbm_terminated0_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TKB0_TKBn}_tkbm_terminated0_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TKB0_TKBn}_tkbm_terminated0_interrupt(void);
```

備考　　*n* = 1, 2　*m* = 0, *n*

[引数]

なし

[戻り値]

なし

r_{Config_TKB0_TKB*n*}_tkb*m*_terminated1_interrupt

TKB*m* TKBO*m*1 の強制出力停止解除割り込み（INTTMKBSTP*m*1）に伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TKB0_TKBn}_tkbm_ terminated1_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TKB0_TKBn}_tkbm_ terminated1_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TKB0_TKBn}_tkbm_ terminated1_interrupt(void);
```

備考　　*n* = 1, 2　*m* = 0, *n*

[引数]

なし

[戻り値]

なし

r_{Config_TKB0_TKB*n*}_tkb*m*_activated0_interrupt

TKB*m* TKBO*m*0 の強制出力停止発動割り込み（INTTMKBSTR*m*0）に伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TKB0_TKBn}_tkbm_activated0_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TKB0_TKBn}_tkbm_activated0_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TKB0_TKBn}_tkbm_activated0_interrupt(void);
```

備考　　*n* = 1, 2　*m* = 0, *n*

[引数]

なし

[戻り値]

なし

r_{Config_TKB0_TKB*n*}_tkb*m*_activated1_interrupt

TKB*m* TKBO*m*1 の強制出力停止発動割り込み（INTTMKBSTR*m*1）に伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TKB0_TKBn}_tkbm_activated1_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TKB0_TKBn}_tkbm_activated1_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TKB0_TKBn}_tkbm_ activated1_interrupt(void);
```

備考　　*n* = 1, 2　*m* = 0, *n*

[引数]

なし

[戻り値]

なし

r_{Config_TKB0_TKB*n*}_tkb*m*_end_count_interrupt

TKB*m* カウント・コンペアー致割り込み（INTTMKBm）の割り込みに伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TKB0_TRD n}_tkb m_end_count _interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TKB0_TRD n}_tkb m_end_count_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TKB0_TRD n}_tkb m_end_count_interrupt(void);
```

備考　　*n* = 1, 2　*m* = 0, *n*

[引数]

なし

[戻り値]

なし

使用例

TKB0 と TKB1 を起動して INTP0 による PWM 波形パルスと強制出力停止機能を出力する例です。
まず、スマート・コンフィグレータで INTP0 を強制出力トリガとして有効にしてください。

main.c

```
#include "r_cg_macrodriver.h"
#include "Config_TKB0_TKB1.h"
#include "Config_INTC.h"

void main(void);

// ミリ秒レベルの遅延をシミュレーションする遅延機能。
static void delay_ms(uint32_t time_ms)
{
	uint32_t i = 0;
	while(time_ms--) {
		for(i = 0; i < 156; i++) {
			NOP();
		}
	}
}

void main(void)
{
	EI();	// 割り込みを有効にします。
	R_Config_TKB0_TKB1_Start();	// TKB0_TKB1 動作を開始します。

	//TKB0 強制出力停止機能 1 を INTP0 で有効にします。
	R_Config_INTC_INTP0_Start();
	delay_ms(10000);
	P14_bit.no0 = ~P14_bit.no0;	// INTP0 へのハイ・レベル信号を生成します。

	// TKB0 の強制出力停止機能 1 を INTP0 で終了します。
	delay_ms(5000);
	P14_bit.no0 = ~P14_bit.no0;	// INTP0 へのロウ・レベル信号を生成します。
	R_Config_TKB0_TKB1_TKBO00_Forced_Output_Stop_Function1_Stop();	// TKBO00 の終了。
	R_Config_TKB0_TKB1_TKBO01_Forced_Output_Stop_Function1_Stop();	// TKBO01 の終了。
	R_Config_INTC_INTP0_Stop();	// INTP0 動作を無効にします。

   while(1);
}
```

4.2.16 PWM 出力（同時スタート／ストップ・モード（TKBCRn0 レジスタによる周期制御）／同時スタート／ストップ・モード（外部トリガ入力による周期制御）／同時スタート／ストップ・モード（マスタによる周期制御）を使用したタイマ KB）（2 スレーブ）

以下に、スマート・コンフィグレータが PWM 出力機能用として出力する API 関数の一覧を示します。

表 4.20 PWM 出力（同時スタート／ストップ・モード（TKBCR*n*0 レジスタによる周期制御）／同時スタート／ストップ・モード（外部トリガ入力による周期制御）／同時スタート／ストップ・モード（マスタによる周期制御）を使用したタイマ KB）用 API 関数（1/2）

API 関数名	周辺機能	機能概要
R_{Config_TKB0_TKB1_TKB2}_Create	タイマ KB	TKB0、TKB1、TKB2 モジュールを同時スタート／ストップ・モード（TKBCR*n*0 レジスタによる周期制御）／同時スタート／ストップ・モード（外部トリガ入力による周期制御）／同時スタート／ストップ・モード（マスタによる周期制御）で制御する前に必要な初期化処理を実行します。
R_{Config_TKB0_TKB1_TKB2}_Start		TKB0、TKB1、TKB2 カウンタを起動します。
R_{Config_TKB0_TKB1_TKB2}_Stop		TKB0、TKB1、TKB2 カウンタを停止します。
R_{Config_TKB0_TKB1_TKB2}_TKB*n*_Set_BatchOverwriteRequestOn		TKB*n* の一斉書き込みを動作します。
R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_Forced_Output_Stop_Function0_Start		TKB*n* TKBO*n*0 の強制出力停止機能 1 を開始します。
R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_Forced_Output_Stop_Function0_Stop		TKB*n* TKBO*n*0 の強制出力停止機能 1 を停止します。
R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_Forced_Output_Stop_Function1_Start		TKB*n* TKBO*n*1 の強制出力停止機能 1 を開始します。
R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_Forced_Output_Stop_Function1_Stop		TKB*n* TKBO*n*1 の強制出力停止機能 1 を停止します。
R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_SmoothStartFunction_Start		TKB*n* TKBO*n*0 のソフト・スタート機能を開始します。
R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_SmoothStartFunction_Stop		TKBn TKBO*n*0 のソフト・スタート機能を停止します。
R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_SmoothStartFunction_Start		TKB*n* TKBO*n*1 のソフト・スタート機能を開始します。
R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_SmoothStartFunction_Stop		TKB*n* TKBO*n*1 のソフト・スタート機能を停止します。

表 4.21 PWM 出力（同時スタート／ストップ・モード（TKBCR*n*0 レジスタによる周期制御）／同時スタート／ストップ・モード（外部トリガ入力による周期制御）／同時スタート／ストップ・モード（マスタによる周期制御）を使用したタイマ KB）用 API 関数（1/2）

API 関数名	周辺機能	機能概要
R_{Config_TKB0_TKB1_TKB2}_Create_UserInit	タイマ KB	TKB0、TKB1、TKB2 に関するユーザ独自の初期化処理を実行します。
r_{Config_TKB0_TKB1_TKB2}_tkbm_terminated0_interrupt		タイマ KB*n* TKBO*n*0 の強制出力停止解除割り込み（INTTMKBSTP*n*0）に伴う処理を実行します。
r_{Config_TKB0_TKB1_TKB2}_tkbm_terminated1_interrupt		タイマ KB*n* TKBO*n*1 の強制出力停止解除割り込み（INTTMKBSTP*n*1）に伴う処理を実行します。
r_{Config_TKB0_TKB1_TKB2}_tkbm_activated0_interrupt		タイマ KB*n* TKBO*n*0 の強制出力停止発動割り込み（INTTMKBSTR*n*0）に伴う処理を実行します。
r_{Config_TKB0_TKB1_TKB2}_tkbm_activated1_interrupt		タイマ KB*n* TKBO*n*1 の強制出力停止発動割り込み（INTTMKBSTR*n*1）に伴う処理を実行します。
r_{Config_TKB0_TKB1_TKB2}_tkbm_end_count_interrupt		タイマ KB*n* カウント・コンペアー致割り込み（INTTMKB*n*）の割り込みに伴う処理を実行します。

R_{Config_TKB0_TKB1_TKB2}_Create

TKB0、TKB1、TKB2 モジュールを同時スタート／ストップ・モード（TKBCRn0 レジスタによる周期制御）／同時スタート／ストップ・モード（外部トリガ入力による周期制御）／同時スタート／ストップ・モード（マスタによる周期制御）で制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、main()関数が実行される前に R_TKB_Create から呼び出されます。

[指定形式]

```
void    R_{Config_TKB0_TKB1_TKB2}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB1_TKB2}_Start

TKB0、TKB1、TKB2 のカウンタを起動します。

[指定形式]

```
void    R_{Config_TKB0_ TKB1_TKB2}_Start(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB1_TKB2}_Stop

TKB0、TKB1、TKB2 カウンタを停止します。

[指定形式]

```
void    R_{Config_TKB0_TKB1_TKB2}_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_ TKB1_TKB2}_TKB*n*_Set_BatchOverwriteRequestOn

TKB*n* の一斉書き込みを動作します。

[指定形式]

```
void    R_{Config_TKB0_ TKB1_TKB2}_TKBn_Set_BatchOverwriteRequestOn (void);
```

備考　　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_Forced_Output_Stop_Function1_Start

TKB0 TKBO*n*0 の強制出力停止機能 1 を開始します。

[指定形式]

void R_{Config_TKB0_ TKB1_TKB2}_TKBO*n*0_Forced_Output_Stop_Function1_Start (void);

備考 *n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_Forced_Output_Stop_Function1_Stop

TKB0 TKBO*n*0 の強制出力停止機能 1 を停止します。

[指定形式]

```
void    R_{Config_TKB0_TKB1_TKB2}_TKBOn0_Forced_Output_Stop_Function1_Stop (void);
```

備考　　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_Forced_Output_Stop_Function1_Start

TKB*n* TKBO*n*1 の強制出力停止機能 1 を開始します。

[指定形式]

```
void    R_{Config_TKB0_TKB1_TKB2}_TKBOn1_Forced_Output_Stop_Function1_Start (void);
```

備考　　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_Forced_Output_Stop_Function1_Stop

TKB*n* と TKBO*n*1 の強制出力停止機能 1 を停止します。

[指定形式]

void R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_Forced_Output_Stop_Function1_Stop (void);

備考 *n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_SmoothStartFunction_Start

TKB*n* と TKBO*n*0 のソフト・スタート機能を開始します。

[指定形式]

void R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_SmoothStartFunction_Start (void);

備考 *n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_SmoothStartFunction_Stop

TKB*n* と TKBO*n*0 のソフト・スタート機能を停止します。

[指定形式]

void R_{Config_TKB0_TKB1_TKB2}_TKBO*n*0_SmoothStartFunction_Stop (void);

備考 *n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_SmoothStartFunction_Start

TKB*n* と TKBO*n*1 のソフト・スタート機能を開始します。

[指定形式]

```
void    R_{Config_TKB0_TKB1_TKB2}_TKBOn1_SmoothStartFunction_Start (void);
```

備考　　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_SmoothStartFunction_Stop

TKB*n* と TKBO*n*1 のソフト・スタート機能を停止します。

[指定形式]

void R_{Config_TKB0_TKB1_TKB2}_TKBO*n*1_SmoothStartFunction_Stop (void);

備考 *n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

R_{Config_TKB0_TKB1_TKB2}_Create_UserInit

TKB0、TKB1、TKB2 に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TKB0_TKB1_TKB2}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TKB0_TKB1_TKB2}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_TKB0_TKB1_TKB2}_tkb*n*_terminated0_interrupt

タイマ KB*n* TKBO*n*0 の強制出力停止解除割り込み（INTTMKBSTP*n*0）に伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TKB0_TKB1_TKB2}_tkbn_terminated0_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TKB0_TKB1_TKB2}_tkbn_terminated0_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TKB0_TKB1_TKB2}_tkbn_terminated0_interrupt(void);
```

備考　　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

r_{Config_TKB0_TKB1_TKB2}_tkb*n*_terminated1_interrupt

タイマ KB*n* TKBO*n*1 の強制出力停止解除割り込み（INTTMKBSTP*n*1）に伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TKB0_TKB1_TKB2}_tkbn_ terminated1_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TKB0_TKB1_TKB2}_tkbn_ terminated1_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TKB0_TKB1_TKB2}_tkbn_ terminated1_interrupt(void);
```

備考　　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

r_{Config_TKB0_TKB1_TKB2}_tkb*n*_activated0_interrupt

タイマ KB*n* TKBO*n*0 の強制出力停止発動割り込み（INTTMKBSTR*n*0）に伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TKB0_TKB1_TKB2}_tkbn_activated0_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TKB0_TKB1_TKB2}_tkbn_activated0_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TKB0_TKB1_TKB2}_tkbn_activated0_interrupt(void);
```

備考　　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

r_{Config_TKB0_TKB1_TKB2}_tkb*n*_activated1_interrupt

タイマ KB*n* TKBO*n*1 の強制出力停止発動割り込み（INTTMKBSTR*n*1）に伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TKB0_TKB1_TKB2}_tkbn_activated1_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TKB0_TKB1_TKB2}_tkbn_activated1_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TKB0_TKB1_TKB2}_tkbn_ activated1_interrupt(void);
```

備考　　*n* = 0, 1, 2

[引数]

なし

[戻り値]

なし

r_{Config_TKB0_TKB1_TKB2}_tkb*n*_end_count_interrupt

タイマ KB*n* カウント・コンペアー致割り込み（INTTMKB*n*）の割り込みに伴う処理を実行します。

［指定形式］

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TKB0_TKB1_TKB2}_tkbn_end_count _interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TKB0_TKB1_TKB2}_tkbn_end_count_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TKB0_TKB1_TKB2}_tkbn_end_count_interrupt(void);
```

備考　　*n* = 0, 1, 2

［引数］

なし

［戻り値］

なし

使用例

TKB0、TKB1、TKB2 を起動して PWM 波形パルスを出力する例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t tkb_pwm_count;

void main(void);

void main(void)
{
    EI();   // 割り込みを有効にします。
    R_Config_TKB0_TKB1_TKB2_Start();   // TKB0、TKB1、TKB2 動作を有効にします。
    while (tkb_pwm_count < 10);     // tkb_pwm_count が 10 以上になるまで待ち、ループを終了します。
    R_Config_TKB0_TKB1_TKB2_Stop();   // TKB0、TKB1、TKB2 動作を無効にします。
}
```

Config_TKB0_TKB1_TKB2_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t tkb_pwm_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TKB0_TKB1_TKB2_tkb0_end_count_interrupt (void)
{
    /* Start user code for r_Config_TKB0_TKB1_TKB2_tkb0_end_count_interrupt. Do not edit comment generated here */
    tkb_pwm_count ++;   // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

4.2.17 入力パルス間隔／周期測定（タイマ・アレイ・ユニット）

以下に、スマート・コンフィグレータが入力パルス間隔測定用として出力する API 関数の一覧を示します。

表 4.22 入力パルス間隔／周期測定（タイマ・アレイ・ユニット）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TAU*m*_*n*}_Create	タイマ・アレイ・ユニット	入力パルス間隔測定モードで TAU*m* チャネル・モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_TAU*m*_*n*}_Start		TAU*m* のチャネル *n* のカウンタを起動します。
R_{Config_TAU*m*_*n*}_Stop		TAU*m* のチャネル *n* のカウンタを停止します。
R_{Config_TAU*m*_*n*}_Get_PulseWidth		TAU*m* のチャネル *n* の入力パルス幅を測定します。
R_{Config_TAU*m*_*n*}_Create_UserInit		TAU*m* のチャネル *n* に関するユーザ独自の初期化処理を実行します。
r_{Config_TAU*m*_*n*}_interrupt		INTTM*mn* 割り込みに伴う処理を実行します。

R_{Config_TAU*m*_*n*}_Create

入力パルス間隔測定モードで TAU*m* チャネル・モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_TAU*m*_Create から呼び出されます。

[指定形式]

```
void    R_{Config_TAUm_n}_Create(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Start

TAU*m* のチャネル *n* のカウンタを起動します。

[指定形式]

```
void    R_{Config_TAUm_n}_Start(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Stop

TAU*m* のチャネル *n* のカウンタを停止します。

[指定形式]

```
void    R_{Config_TAUm_n}_Stop(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Get_PulseWidth

TAU*m* のチャネル *n* の入力パルス幅を測定します。

[指定形式]

void R_{Config_TAU*m*_*n*}_Get_PulseWidth(uint32_t * const width);

備考 *m* はユニット番号、*n* はチャネル番号を示します。

[引数]

I/O	引数	説明
O	uint32_t * const width;	入力パルス幅を格納するアドレス

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Create_UserInit

TAU*m* のチャネル *n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TAU*m*_*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TAUm_n}_Create_UserInit(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_TAU*m*_*n*}_interrupt

INTTM*mn* 割り込みに伴う処理を実行します。

備考　この API 関数は、キャプチャ割り込みの割り込みハンドラ（INTTM*mn*）として呼ばれ、有効なキャプチャ・エッジが検出され、カウンタ値（TCR*mn*）がタイマ・データ・レジスタ mn（TDR*mn*）に転送されたときに発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TAUm_n}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TAUm_n}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TAUm_n}_interrupt(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

使用例

TI00 から TAU チャネル 0 の入力間隔幅を取得するための例です。

main.c

```
#include "r_smc_entry.h"
extern volatile uint8_t tau_interrupt_flag;
uint32_t width;

void main(void);

void main(void)
{
    EI();    // 割り込みを有効にします。
    tau_interrupt_flag = 0;    // tau_interrupt_flag の初期値を設定します。
    R_Config_TAU0_0_Start();    // TAU00 動作を有効にすると、TCR00 レジスタは 0000H からカウントアップされます。
    while( tau_interrupt_flag == 0);
    R_Config_TAU0_0_Stop();    // TAU00 動作を無効にします。
    // TI00 端子のハイ・レベル幅またはロウ・レベル幅は、「width」 から読み取ることができます。
    R_Config_TAU0_0_Get_PulseWidth(&width);
}
```

Config_TAU0_0_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t tau_interrupt_flag;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_0_interrupt (void)
{
    ...
    /* Start user code for r_Config_TAU0_0_interrupt. Do not edit comment generated here */
    // TI00 端子入力の有効なエッジが検出され、カウント値が TDR00 に転送されます。
    tau_interrupt_flag = 1U;    /* Set the flag */
    /* End user code. Do not edit comment generated here */
}
```

4.2.18 入力パルス間隔／周期測定（タイマ RJ）

以下に、スマート・コンフィグレータが入力パルス間隔（TRJIO*n* 端子への入力）測定用として出力する API 関数の一覧を示します。

表 4.23 入力パルス間隔／周期測定（タイマ RJ）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TRJ*n*}_Create	タイマ RJ	入力パルス幅測定モードで TRJ*n* モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_TRJ*n*}_Start		TRJ*n* のカウンタを起動します。
R_{Config_TRJ*n*}_Stop		TRJ*n* のカウンタを停止します。
R_{Config_TRJ*n*}_Get_PulseWidth		TRJ*n* のチャネル *n* の入力パルス幅を測定します。
R_{Config_TRJ*n*}_Create_UserInit		TRJ*n* に関するユーザ独自の初期化処理を実行します。
r_{Config_TRJ*n*}_interrupt		INTTRJ*n* 割り込みに伴う処理を実行します。

R_{Config_TRJ*n*}_Create

外部信号（TRJIO*n* 端子への入力）測定モードの入力パルス幅で TRJ*n* モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_TRJn}_Create(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TRJ*n*}_Start

TRJ*n* のカウンタを起動します。

[指定形式]

```
void    R_{Config_TRJn}_Start(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TRJ*n*}_Stop

TRJ*n* のカウンタを停止します。

[指定形式]

```
void    R_{Config_TRJn}_Stop(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TRJ*n*}_Get_PulseWidth

TRJ*n* の入力パルス幅を測定します。

[指定形式]

void R_{Config_TRJ*n*}_Get_PulseWidth(uint32_t * const width);

備考 *n* はチャネル番号を示します。

[引数]

I/O	引数	説明
O	uint32_t * const width;	入力パルス幅を格納するアドレス

[戻り値]

なし

R_{Config_TRJ*n*}_Create_UserInit

TRJ*n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TRJ*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TRJn}_Create_UserInit(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_TRJ*n*}_interrupt

INTTRJ*n* 割り込みに伴う処理を実行します。

備考　この API 関数は、パルス幅測定モードで外部入力（TRJIO*n*）のアクティブ幅測定が完了したときに発生するキャプチャ割り込み（INTTRJ*n*）の割り込みハンドラとして呼び出されます。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TRJn}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TRJn}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TRJn}_interrupt(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

使用例

TRJ0 の入力間隔幅を取得するための例です。

main.c

```
#include "r_smc_entry.h"
volatile uint8_t measure_flag = 0U;
uint32_t g_width[20] = {0};

void main(void);

void main(void)
{
    EI();    // 割り込みを有効にします。
    R_Config_TRJ0_Start();    // TRJ0 動作が有効になると、TCR00 レジスタは 0000H からカウントアップされます。
    R_Config_TAU0_0_Start();  // TAU0_0 から方形波を出力します。
    // 最初の割り込みはダミー値で、g_width に格納する必要はありません。
    while(measure_flag != 1U);
    measure_flag = 0U;

    for (char i = 0; i < 20; i++)    // 20 回測定します。
    {
        while(measure_flag != 1U);    // measure_flag が 1 になるまで待ってから、ループを終了します。TRJIO0 端子入力の有効なエッジが検出されたことを意味します。
        R_Config_TRJ0_Get_PulseWidth(g_width + i);    // TRJIO0 端子のパルス周期を取得すると、g_width から読み取ることができます。
        measure_flag = 0U;
    }
    R_Config_TRJ0_Stop();    // TRJ0 動作を無効にします。
    while (1U);
}
```

Config_TRJ0_0_user.c

```
/* Start user code for global. Do not edit comment generated here */
extern volatile uint8_t measure_flag;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TRJ0_interrupt (void)
{
    ...
    /* Start user code for r_Config_TRJ0_interrupt. Do not edit comment generated here */
    measure_flag = 1U;    // 外部入力の有効幅 (TRJIO0) の測定が完了した場合。
    /* End user code. Do not edit comment generated here */
}
```

4.2.19 インターバル・タイマ（タイマ・アレイ・ユニット）

以下に、スマート・コンフィグレータがインターバル・タイマ（タイマ・アレイ・ユニット）用として出力する API 関数の一覧を示します。

表 4.24 インターバル・タイマ（タイマ・アレイ・ユニット）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TAU*m*_*n*}_Create	タイマ・アレイ・ユニット	TAU*m* チャネル・モジュールをインターバル・タイマ・モードで制御する前に必要な初期化処理を実行します。
R_{Config_TAU*m*_*n*}_Start		TAU*m* のチャネル *n* のカウンタを起動します。
R_{Config_TAU*m*_*n*}_Stop		TAU*m* のチャネル *n* のカウンタを停止します。
R_{Config_TAU*m*_*n*}_Higher8bits_Start		TAU*m* のチャネル *n* の上位側 8 ビット・カウンタを起動します。
R_{Config_TAU*m*_*n*}_Higher8bits_Stop		TAU*m* のチャネル *n* の上位側 8 ビット・カウンタを停止します。
R_{Config_TAU*m*_*n*}_Lower8bits_Start		TAU*m* のチャネル *n* の下位側 8 ビット・カウンタを起動します。
R_{Config_TAU*m*_*n*}_Lower8bits_Stop		TAU*m* のチャネル *n* の下位側 8 ビット・カウンタを停止します。
R_{Config_TAU*m*_*n*}_Create_UserInit		TAU*m* のチャネル *n* に関するユーザ独自の初期化処理を実行します。
r_{Config_TAU*m*_*n*}_interrupt		INTTM*mn* 割り込みに伴う処理を実行します。
r_{Config_TAU*m*_*n*}_higher8bits_interrupt		INTTM*mn*H 割り込みに伴う処理を実行します。 (上位 8 ビット・タイマ動作時)

R_{Config_TAU*m*_*n*}_Create

TAU*m* チャネル・モジュールをインターバル・タイマ・モードで制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_TAU*m*_Create から呼び出されます。

[指定形式]

```
void    R_{Config_TAUm_n}_Create(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Start

TAU*m* のチャネル *n* のカウンタを起動します。

[指定形式]

```
void    R_{Config_TAUm_n}_Start(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Stop

TAU*m* のチャネル *n* のカウンタを停止します。

[指定形式]

```
void    R_{Config_TAUm_n}_Stop(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Higher8bits_Start

TAU*m* のチャネル *n* の上位側 8 ビット・カウンタを起動します。

[指定形式]

```
void    R_{Config_TAUm_n}_Higher8bits_Start(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号 1 または 3 を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Higher8bits_Stop

TAU*m* のチャネル *n* の上位側 8 ビット・カウンタを停止します。

[指定形式]

void R_{Config_TAU*m*_*n*}_Higher8bits_Stop(void);

備考 *m* はユニット番号、*n* はチャネル番号 1 または 3 を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Lower8bits_Start

TAU*m* のチャネル *n* の下位側 8 ビット・カウンタを起動します。

[指定形式]

```
void    R_{Config_TAUm_n}_Lower8bits_Start(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号 1 または 3 を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Lower8bits_Stop

TAU*m* のチャネル *n* の下位側 8 ビット・カウンタを停止します。

[指定形式]

```
void    R_{Config_TAUm_n}_Lower8bits_Stop(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号 1 または 3 を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Create_UserInit

TAU*m* のチャネル *n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TAU*m*_*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TAUm_n}_Create_UserInit(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_TAU*m*_*n*}_interrupt

INTTM*mn* 割り込みに伴う処理を実行します。

備考　この API 関数は、カウント終了割り込み（INTTM*mn*）の割り込みハンドラとして呼び出され、カウンタ値（TCR*mn*）が 0000H に達したときに発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TAUm_n}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TAUm_n}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TAUm_n}_interrupt(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_TAU*m*_*n*}_higher8bits_interrupt

INTTM*mn*H 割り込みに伴う処理を実行します。（上位 8 ビット・タイマ動作時）

備考　この API 関数は、カウント終了割り込み（INTTM*mn*）の割り込みハンドラとして呼び出され、カウンタ（TCR*mn*）の上位 8 ビット値が 00H に達したときに発生します。

［指定形式］

CCRL78 ツールチェーンの場合

```
static void __near   r_{Config_TAUm_n}_higher8bits_interrupt(void);
```

LLVM ツールチェーンの場合

```
void   r_{Config_TAUm_n}_higher8bits_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void   r_{Config_TAUm_n}_higher8bits_interrupt(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

［引数］

なし

［戻り値］

なし

使用例

TAU チャネル 0 カウントのインターバル・タイマを使用し、チャネル 3 を上位 8 ビット・インターバル・タイマとしてカウントし、チャネル 1 を下位 8 ビット・インターバル・タイマとしてカウントする例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t ch0_run_count;
extern uint8_t ch1_run_count;
extern uint8_t ch3_run_count;

void main(void);

void main(void)
{
    EI();    // 割り込みを有効にします。
    R_Config_TAU0_0_Start();    // TAU00 動作を有効にします。
    while (ch0_run_count < 20);    // ch0_run_count が 20 以上になるまで待ち、ループを終了します。
    R_Config_TAU0_0_Stop();    // TAU00 動作を無効にします。

    R_Config_TAU0_1_Lower8bits_Start();    // TAU01 動作を有効にします。
    while (ch1_run_count < 20);    // ch1_run_count が 20 以上になるまで待ち、ループを終了します。
    R_Config_TAU0_1_Lower8bits_Stop();    // TAU01 動作を無効にします。

    R_Config_TAU0_3_Higher8bits_Start();    // TAU03 動作を有効にします。
    while (ch3_run_count < 20);    // ch3_run_count が 20 以上になるまで待ち、ループを終了します。
    R_Config_TAU0_3_Higher8bits_Stop();    // TAU03 動作を無効にします。
}
```

Config_TAU0_0_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch0_run_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_0_interrupt (void)
{
    /* Start user code for r_Config_ TAU0_0_interrupt. Do not edit comment generated here */
    ch0_run_count ++;    // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

Config_TAU0_1_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch1_run_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_1_interrupt (void)
{
    /* Start user code for r_Config_TAU0_1_interrupt. Do not edit comment generated here */
    ch1_run_count ++;    // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

Config_TAU0_3_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch3_run_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_3 _interrupt (void)
{
    /* Start user code for r_Config_TAU0_3 _interrupt. Do not edit comment generated here */
      ch3_run_count ++;    // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

4.2.20 インターバル・タイマ（タイマ RJ）

以下に、スマート・コンフィグレータがインターバル・タイマ（タイマ RJ*n*）用として出力する API 関数の一覧を示します。

表 4.25 インターバル・タイマ（タイマ RJ*n*）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TRJ*n*}_Create	タイマ RJ*n*	TRJ*n* モジュールをインターバル・タイマ・モードで制御する前に必要な初期化処理を実行します。
R_{Config_TRJ*n*}_Start		TRJ*n* のカウンタを起動します。
R_{Config_TRJ*n*}_Stop		TRJ*n* のカウンタを停止します。
R_{Config_TRJ*n*}_Create_UserInit		TRJ*n* に関するユーザ独自の初期化処理を実行します。
r_{Config_TRJ*n*}_interrupt		INTTR*n* 割り込みに伴う処理を実行します。

R_{Config_TRJ*n*}_Create

TRJ*n* モジュールをインターバル・タイマ・モードで制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_TRJn}_Create(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TRJ*n*}_Start

TRJ*n* のカウンタを起動します。

[指定形式]

```
void    R_{Config_TRJn}_Start(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TRJ*n*}_Stop

TRJ*n* のカウンタを停止します。

[指定形式]

```
void    R_{Config_TRJn}_Stop(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TRJ*n*}_Create_UserInit

TRJ*n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TRJ*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TRJn}_Create_UserInit(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_TRJ*n*}_interrupt

INTTRJ*n* 割り込みに伴う処理を実行します。

備考　この API 関数は、TRJ*n* 割り込み（INTTRJ*n*）の割り込みハンドラとして呼び出され、カウンタ値が 0000H に達し、次のカウント・ソースが入力されたときに発生し、カウンタがアンダフローになります。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TRJn}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TRJn}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TRJn}_interrupt(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

使用例

ユーザ定義のカウンタ値に TRJ0 カウントを使用し、波形 P00 端子に出力する例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t ch0_run_count

void main(void);

void main(void)
{
    EI();    // 割り込みを有効にします。
    R_Config_TRJ0_Start();    // TRJ0 動作を有効にします。
    while (ch0_run_count < 20);    // ch0_run_count が 20 以上になるまで待ち、ループを終了します。
    R_Config_TRJ0_Stop();    // TRJ0 動作を無効にします。
}
```

Config_TRJ0_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch0_run_count
/* End user code. Do not edit comment generated here */

static void __near r_Config_TRJ0_interrupt (void)
{
    /* Start user code for r_Config_TRJ0_interrupt. Do not edit comment generated here */
    ch0_run_count ++;    // 割り込みハンドラが入力された回数をカウントします。
    P0_bit.no0 = ~P0_bit.no0;    // 割り込みが発生するたびに、P0.0 のレベルが 1 回反転します。
    /* End user code. Do not edit comment generated here */
}
```

4.2.21 インターバル・タイマ（12 ビット・インターバル・タイマ）

以下に、スマート・コンフィグレータがインターバル・タイマ（12 ビット・インターバル・タイマ）用として出力する API 関数の一覧を示します。

表 4.26 インターバル・タイマ（12 ビット・インターバル・タイマ）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_IT}_Create	12 ビット・インターバル・タイマ	12 ビット・インターバル・タイマ・モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_IT}_Start		12 ビット・インターバル・タイマのカウンタを起動します。
R_{Config_IT}_Stop		12 ビット・インターバル・タイマのカウンタを停止します。
R_{Config_IT}_Create_UserInit		12 ビット・インターバル・タイマに関するユーザ独自の初期化処理を実行します。
r_{Config_IT}_interrupt		INTIT 割り込みに伴う処理を実行します。

R_{Config_IT}_Create

12 ビット・インターバル・タイマ・モジュールをインターバル・タイマ・モードで制御する前に必要な初期化処理を実行します。

備考　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_IT}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_IT}_Start

IT カウンタを起動します。

[指定形式]

```
void    R_{Config_IT}_Start(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_IT}_Stop

IT カウンタを停止します。

[指定形式]

```
void    R_{Config_IT}_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_IT_Create_UserInit

12 ビット・インターバル・タイマに関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_IT}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_IT}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_IT}_interrupt

12 ビット・インターバル・タイマカウント値がコンペア値に達する時、INTIT 割り込みに伴う処理を実行します。

備考 この API 関数は、12 ビット・インターバル・タイマ割り込み (INTIT) の割り込みハンドラとして呼び出され、カウント値がコンペア値に達すると発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_IT}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_IT}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_IT}_interrupt(void);
```

[引数]

なし

[戻り値]

なし

使用例

ユーザ定義のカウンタ値に 12 ビット・インターバル・タイマ・カウントを使用し、P00 端子から波形を出力する例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t count;

void main(void);

void main(void)
{
    EI();    // 割り込みを有効にします。
    R_Config_IT_Start();    // IT 動作を有効にします。
    while (count < 20);   // count が 20 以上になるまで待ち、ループを終了します。
    R_Config_IT_Stop();    // IT 動作を無効にします。
}
```

Config_IT_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_IT_interrupt (void)
{
    /* Start user code for r_Config_IT_interrupt. Do not edit comment generated here */
    count ++;    // 割り込みハンドラが入力された回数をカウントします。
    P0_bit.no0 = ~P0_bit.no0;    // 割り込みが発生するたびに、P00 のレベルが 1 回反転します。
    /* End user code. Do not edit comment generated here */
}
```

4.2.22 インターバル・タイマ（8 ビット・カウント・モードを使用した 32 ビット・インターバル・タイマ）

以下に、スマート・コンフィグレータが 32 ビット・インターバル・タイマの 8 ビット・カウント・モード用として出力する API 関数の一覧を示します。

表 4.27 インターバル・タイマ（8 ビット・カウント・モードを使用した 32 ビット・インターバル・タイマ）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_ITL*n*}_Create	32 ビット・インターバル・タイマ	32 ビット・インターバル・タイマ（8 ビット・モード）で ITL*n* モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_ITL*n*}_Start		ITL*n* チャネルを起動します。
R_{Config_ITL*n*}_Stop		ITL*n* チャネルを停止します。
R_{Config_ITL*n*}_Set_OperationMode		32 ビット・インターバル・タイマの動作モードを変更する前に、カウンタを停止して割り込みフラグをクリアするために使用します。
R_{Config_ITL*n*}_Create_UserInit		ITL*n* チャネルに関するユーザ独自の初期化処理を実行します。
r_{Config_ITL*n*}_Callback_Shared_interrupt		32 ビット・インターバル・タイマ割り込み（INTITL）に伴う処理を実行します。

R_{Config_ITL*n*}_Create

32 ビット・インターバル・タイマの 8 ビット・モードで ITL*n* モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_ITL_Create から呼び出されます。

[指定形式]

```
void    R_{Config_ITLn}_Create(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_ITL*n*}_Start

ITL*n* チャネルを起動します。

備考　32 ビット・インターバル・タイマ割り込みは、R_ITL_Start_Interrupt()で呼び出すことで有効になります。32 ビット・インターバル・タイマ割り込みを使用する場合は、R_ITL_Start_Interrupt()と共にこの API 関数を呼び出してください。

[指定形式]

```
void    R_{Config_ITLn}_Start(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_ITL*n*}_Stop

ITL*n* チャネルを停止します。

[指定形式]

```
void    R_{Config_ITLn}_Stop(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_ITL*n*}_Set_OperationMode

32 ビット・インターバル・タイマの動作モードを変更する前に、カウンタを停止して割り込みフラグをクリアするために使用します。

[指定形式]

```
void    R_{Config_ITLn}_Set_OperationMode(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_ITL*n*}_Create_UserInit

ITL*n* チャネルに関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_ITL*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_ITLn}_Create_UserInit(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_ITL*n*}_Callback_Shared_interrupt

32 ビット・インターバル・タイマ割り込み（INTITL）に伴う処理を実行します。

備考 1　この API 関数は、32 ビット・インターバル・タイマ割り込みの割り込みハンドラである r_itl_interrupt からコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアのみを保持する必要があり、他の処理コードは、コールバックおよび割り込み関数から移動する必要があります。それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_ITLn}_Callback_Shared_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_ITLn}_Callback_Shared_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_ITLn}_Callback_Shared_interrupt(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

使用例

ユーザ定義周期に 8 ビット・カウント・モードを開始するための例です。

main.c

```
#include "r_smc_entry.h"
extern volatile uint8_t interrupt_flag;

void main(void);

void main(void)
{
    EI();   // 割り込みを有効にします。
    interrupt_flag = 0;
    R_ITL_Start_Interrupt();
    R_Config_ITL001 _Start();   // ITL001 動作を有効にします。
    while (interrupt_flag  < 20);   // interrupt_flag が 20 以上になるまで待ち、ループを終了します。
    R_Config_ITL001 _Stop();   // ITL001 動作を無効にします。
}
```

Config_ITL000_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t interrupt_flag;
/* End user code. Do not edit comment generated here */

void R_Config_ITL000_Callback_Shared_Interrupt(void)
{
    /* Start user code for R_Config_ITL000_Callback_Shared_Interrupt. Do not edit comment generated here */
    interrupt_flag ++;   // 割り込みハンドラが入力された回数をカウントします。
    P0 = ~P0;   // 割り込みが発生するたびに、P00 のレベルが 1 回反転します。
    /* End user code. Do not edit comment generated here */
}
```

4.2.23 インターバル・タイマ（16 ビット・カウント・モードを使用した 32 ビット・インターバル・タイマ）

以下に、スマート・コンフィグレータが 32 ビット・インターバル・タイマの 16 ビット・カウント・モード用として出力する API 関数の一覧を示します。

表 4.28 インターバル・タイマ（16 ビット・カウント・モードを使用した 32 ビット・インターバル・タイマ）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_ITL*n*_ITL*m*}_Create	32 ビット・インターバル・タイマ	32 ビット・インターバル・タイマ（16 ビット・モード）で ITL*n*_ITL*m* モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_ITL*n*_ITL*m*}_Start		ITL*n*_ITL*m* チャネルを起動します。
R_{Config_ITL*n*_ITL*m*}_Stop		ITL*n*_ITL*m* チャネルを停止します。
R_{Config_ITL*n*_ITL*m*}_Set_SoftwareTriggerOn		ソフトウェア・トリガを発生させます。
R_{Config_ITL*n*_ITL*m*}_Set_OperationMode		32 ビット・インターバル・タイマの動作モードを変更する前に、カウンタを停止して割り込みフラグをクリアするために使用します。
R_{Config_ITL*n*_ITL*m*}_Get_CaptureValue		キャプチャ値を取得します。
R_{Config_ITL*n*_ITL*m*}_Create_UserInit		ITL*n*_ITL*m* チャネルに関するユーザ独自の初期化処理を実行します。
r_{Config_ITL*n*_ITL*m*}_Callback_Shared_interrupt		32 ビット・インターバル・タイマ割り込み（INTITL）に伴う処理を実行します。

R_{Config_ITL*n*_ITL*m*}_Create

32 ビット・インターバル・タイマ（16 ビット・モード）で ITL*n*_ITL*m* モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_ITL_Create から呼び出されます。

[指定形式]

```
void    R_{Config_ITLn_ITLm}_Create(void);
```

備考　*n* が 000 のとき、*m* は 001 です。*n* が 012 のとき、m は 013 です。

[引数]

なし

[戻り値]

なし

R_{Config_ITL*n*_ITL*m*}_Start

ITL*n*_ITL*m* チャネルを起動します。

備考　32 ビット・インターバル・タイマ割り込みは、R_ITL_Start_Interrupt()で呼び出すことで有効になります。32 ビット・インターバル・タイマ割り込みを使用する場合は、R_ITL_Start_Interrupt()と共にこの API 関数を呼び出してください。

[指定形式]

```
void    R_{Config_ITLn_ITLm}_Start(void);
```

備考　*n* が 000 のとき、*m* は 001 です。*n* が 012 のとき、m は 013 です。

[引数]

なし

[戻り値]

なし

R_{Config_ITL*n*_ITL*m*}_Stop

ITL*n*_ITL*m* チャネルを停止します。

[指定形式]

```
void    R_{Config_ITLn_ITLm}_Stop(void);
```

備考　　*n* が 000 のとき、*m* は 001 です。*n* が 012 のとき、m は 013 です。

[引数]

なし

[戻り値]

なし

R_{Config_ITL*n*_ITL*m*}_Set_SoftwareTriggerOn

ソフトウェア・トリガを発生させます。

[指定形式]

void R_{Config_ITL*n*_ITL*m*}_Set_SoftwareTriggerOn(void);

備考 *n* が 000 のとき、*m* は 001 です。*n* が 012 のとき、m は 013 です。

[引数]

なし

[戻り値]

なし

R_{Config_ITL*n*_ITL*m*}_Set_OperationMode

32 ビット・インターバル・タイマの動作モードを変更する前に、カウンタを停止して割り込みフラグをクリアするために使用します。

[指定形式]

```
void    R_{Config_ITLn_ITLm}_Set_OperationMode(void);
```

備考　　*n* が 000 のとき、*m* は 001 です。*n* が 012 のとき、m は 013 です。

[引数]

なし

[戻り値]

なし

R_{Config_ITL*n*_ITL*m*}_Get_CaptureValue

キャプチャ値を取得します。

[指定形式]

```
void    R_{Config_ITLn_ITLm}_Get_CaptureValue(uint16_t * const value);
```

備考　　*n* が 000 のとき、*m* は 001 です。*n* が 012 のとき、m は 013 です。

[引数]

I/O	引数	説明
O	uint16_t * const value;	キャプチャ値を格納するアドレス

[戻り値]

なし

R_{Config_ITL*n*_ITL*m*}_Create_UserInit

ITL*n*_ITL*m* チャネルに関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_ITL*n*_ITL*m*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_ITLn_ITLm}_Create_UserInit(void);
```

備考　*n* が 000 のとき、*m* は 001 です。*n* が 012 のとき、m は 013 です。

[引数]

なし

[戻り値]

なし

r_{Config_ITL*n*_ITL*m*}_Callback_Shared_interrupt

32 ビット・インターバル・タイマ割り込み（INTITL）に伴う処理を実行します。

備考 1　この API 関数は、32 ビット・インターバル・タイマ割り込みの割り込みハンドラである r_itl_interrupt からコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアのみを保持する必要があり、他の処理コードは、コールバックおよび割り込み関数から移動する必要があります。それ以外は、割り込みが正しいタイミングで処理されません。

［指定形式］

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_ITLn_ITLm}_Callback_Shared_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_ITLn_ITLm}_Callback_Shared_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_ITLn_ITLm}_Callback_Shared_interrupt(void);
```

備考　*n* が 000 のとき、*m* は 001 です。*n* が 012 のとき、m は 013 です。

［引数］

なし

［戻り値］

なし

使用例

32 ビット・インターバル・タイマの動作モードを変更する例です（16 ビット・カウント・モードから 16 ビット・キャプチャ・モードに変更します）。

main.c

```
#include "r_smc_entry.h"

void main(void);

void main(void)
{
    EI();   // 割り込みを有効にします。
    R_Config_ITL000_ITL001_Set_OperationMode();     // 32 ビット・インターバル・タイマの全てのカウンタを無効にします。
    /* Capture setting */
    ITLCC0 |= _80_ITL_CAPTURE_ENABLE;
    CAPF0CR = 1U;
    ITLCC0 |= _00_ITL_CAPTURE_COUNTER_RETAIN;
    ITLCC0 &= _FC_ITL_CAPTURE_TRIGGER_CLEAR;
    ITLCC0 |= _00_ITL_CAPTURE_TRIGGER_SOFTWARE;
    R_Config_ITL000_ITL0011_Start();   // ITL00_ITL001 動作を有効にします。
    R_Config_ITL000_ITL0011_Set_SoftwareTriggerOn();   // キャプチャ用のソフトウェア・トリガを生成します。
}
```

Config_ITL000_user.c

```
volatile uint16_t value = 0U;

void r_Config_ITL000_callback_shared_interrupt(void)
{
    // キャプチャ・トリガを検出すると、ITLCAP00 レジスタは値から読み込むことができます。
    R_Config_ITL000_ITL0011_Get_CaptureValue (&value);
}
```

4.2.24 インターバル・タイマ（32 ビット・カウント・モードを使用した 32 ビット・インターバル・タイマ）

以下に、スマート・コンフィグレータが 32 ビット・インターバル・タイマの 32 ビット・カウント・モード用として出力する API 関数の一覧を示します。

表 4.29 インターバル・タイマ（32 ビット・カウント・モードを使用した 32 ビット・インターバル・タイマ）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_ITL000_ITL001_ITL012_ITL013}_Create	32 ビット・インターバル・タイマ	32 ビット・インターバル・タイマ（32 ビット・モード）で ITL000_ITL001_ITL012_ITL013 モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_ITL000_ITL001_ITL012_ITL013}_Start		ITL000_ITL001_ITL012_ITL013 チャネルを起動します。
R_{Config_ITL000_ITL001_ITL012_ITL013}_Stop		ITL000_ITL001_ITL012_ITL013 チャネルを停止します。
R_{Config_ITL000_ITL001_ITL012_ITL013}_Set_OperationMode		32 ビット・インターバル・タイマの動作モードを変更する前に、カウンタを停止して割り込みフラグをクリアするために使用します。
R_{Config_ITL000_ITL001_ITL012_ITL013}_Create_UserInit		ITL000_ITL001_ITL012_ITL013 チャネルに関するユーザ独自の初期化処理を実行します。
r_{Config_ITL000_ITL001_ITL012_ITL013}_Callback_Shared_Interrupt		32 ビット・インターバル・タイマ割り込み（INTITL）に伴う処理を実行します。

R_{Config_ITL000_ITL001_ITL012_ITL013}_Create

32 ビット・インターバル・タイマ（32 ビット・モード）で ITL000_ITL001_ITL012_ITL013 モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_ITL_Create から呼び出されます。

[指定形式]

```
void    R_{Config_ITL000_ITL001_ITL012_ITL013}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_ITL000_ITL001_ITL012_ITL013}_Start

ITL000_ITL001_ITL012_ITL013 チャネルを起動します。

備考　32 ビット・インターバル・タイマ割り込みは、R_ITL_Start_Interrupt()で呼び出すことで有効になります。32 ビット・インターバル・タイマ割り込みを使用する場合は、R_ITL_Start_Interrupt()と共にこの API 関数を呼び出してください。

[指定形式]

```
void    R_{Config_ITL000_ITL001_ITL012_ITL013}_Start(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_ITL000_ITL001_ITL012_ITL013}_Stop

ITL000_ITL001_ITL012_ITL013 チャネルを停止します。

[指定形式]

```
void    R_{Config_ITL000_ITL001_ITL012_ITL013}_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_ITL000_ITL001_ITL012_ITL013}_Set_OperationMode

32 ビット・インターバル・タイマの動作モードを変更する前に、カウンタを停止して割り込みフラグをクリアするために使用します。

[指定形式]

```
void    R_{Config_ITL000_ITL001_ITL012_ITL013}_Set_OperationMode(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_ITL000_ITL001_ITL012_ITL013}_Create_UserInit

ITL000_ITL001_ITL012_ITL013 チャネルに関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_ITL000_ITL001_ITL012_ITL013}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_ITL000_ITL001_ITL012_ITL013}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_ITL000_ITL001_ITL012_ITL013}_Callback_Shared_interrupt

32 ビット・インターバル・タイマ割り込み（INTITL）に伴う処理を実行します。

備考 1　この API 関数は、32 ビット・インターバル・タイマ割り込みの割り込みハンドラである r_itl_interrupt からコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアのみを保持する必要があり、他の処理コードは、コールバックおよび割り込み関数から移動する必要があります。それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_ITL000_ITL001_ITL012_ITL013}_Callback_Shared_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_ITL000_ITL001_ITL012_ITL013}_Callback_Shared_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void
r_{Config_ITL000_ITL001_ITL012_ITL013}_Callback_Shared_interrupt(void);
```

[引数]

なし

[戻り値]

なし

使用例

32 ビット・カウント・モードの使用例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t itl_run_count

void main(void);

void main(void)
{
    EI();    // 割り込みを有効にします。
    R_Config_ITL000_ITL001_ITL012_ITL013_Start();    // ITL00 から ITL003 動作を有効にします。
    while (itl_run_count < 20);    // itl_run_count が 20 以上になるまで待ち、ループを終了します。
    R_Config_ITL000_ITL001_ITL012_ITL013_Stop();    // ITL00 から ITL003 動作を無効にします。
}
```

Config_ITL000_ITL001_ITL012_ITL013_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t itl_run_count;
/* End user code. Do not edit comment generated here */

void R_Config_ITL000_ITL001_ITL012_ITL013_Callback_Shared_Interrupt(void)
{
    /* Start user code for R_Config_ITL000_ITL001_ITL012_ITL013_Callback_Shared_Interrupt. Do not edit comment generated here */
    itl_run_count ++;    // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

4.2.25 インターバル・タイマ（8 ビット・カウント・モードを使用した 8 ビット・インターバル・タイマ）

以下に、スマート・コンフィグレータが 8 ビット・インターバル・タイマの 8 ビット・カウント・モード用として出力する API 関数の一覧を示します。

表 4.30 インターバル・タイマ（8 ビット・カウント・モードを使用した 8 ビット・インターバル・タイマ）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_IT*mn*}_Create	8 ビット・インターバル・タイマ	8 ビット・インターバル・タイマ・モードで IT*mn* モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_IT*mn*}_Start		IT*mn* チャネルを起動します。
R_{Config_IT*mn*}_Stop		IT*mn* チャネルを停止します。
R_{Config_IT*mn*}_Create_UserInit		IT*mn* チャネルに関するユーザ独自の初期化処理を実行します。
r_{Config_IT*mn*}_interrupt		8 ビット・インターバル・タイマ・モード割り込み（INTIT*mn*）に伴う処理を実行します。

R_{Config_ITmn}_Create

8 ビット・インターバル・タイマ・モードで ITmn モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_ITm_Create から呼び出されます。

[指定形式]

```
void    R_{Config_ITmn}_Create(void);
```

備考　m はユニット番号、n はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_IT*mn*}_Start

IT*mn* チャネルを起動します。

[指定形式]

```
void    R_{Config_ITmn}_Start(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_IT*mn*}_Stop

IT*mn* チャネルを停止します。

[指定形式]

void R_{Config_IT*mn*}_Stop(void);

備考 *m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_IT*mn*}_Create_UserInit

IT*mn* チャネルに関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_IT*mn*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_ITmn}_Create_UserInit(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_IT*mn*}_interrupt

8 ビット・インターバル・タイマ・モード割り込み（INTIT*mn*）に伴う処理を実行します。

備考　この API 関数は、カウント終了割り込み（INTIT*mn*）の割り込みハンドラとして呼び出され、カウンタ値（TRTCMP*mn*）が 00H に達したときに発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_IT mn}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_IT mn}_Callback_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_IT mn}_interrupt(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

使用例

ユーザ定義のカウンタ値に 8 ビット・カウントを使用し、P00 から波形を出力する例です。

main.c

```
#include "r_smc_entry.h"
extern volatile uint8_t interrupt_flag;

void main(void);

void main(void)
{
    EI();    // 割り込みを有効にします。
    interrupt_flag = 0;
    R_Config_IT01_Start();    // IT01 動作を有効にします。
    while (interrupt_flag < 20);    // interrupt_flag が 20 以上になるまで待ち、ループを終了します。
    R_Config_IT01_Stop();    // IT01 動作を無効にします。
}
```

Config_IT01_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t interrupt_flag;
/* End user code. Do not edit comment generated here */

void R_Config_IT01_Interrupt(void)
{
    /* Start user code for R_Config_IT01_Interrupt. Do not edit comment generated here */
    interrupt_flag ++;    // 割り込みハンドラが入力された回数をカウントします。
    P0_bit.no0 = ~P0_bit.no0;    // 割り込みが発生するたびに、P00 のレベルが 1 回反転します。
    /* End user code. Do not edit comment generated here */
}
```

4.2.26 インターバル・タイマ（16 ビット・カウント・モードを使用した 8 ビット・インターバル・タイマ）

以下に、スマート・コンフィグレータが 8 ビット・インターバル・タイマの 16 ビット・カウント・モード用として出力する API 関数の一覧を示します。

表 4.31 インターバル・タイマ（16 ビット・カウント・モードを使用した 8 ビット・インターバル・タイマ）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_IT*m*0_IT*m*1}_Create	8 ビット・インターバル・タイマ	16 ビット・インターバル・タイマ・モードで IT*m* モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_IT*m*0_IT*m*1}_Start		IT*m* ユニットを起動します。
R_{Config_ IT*m*0_IT*m*1}_Stop		IT*m* ユニットを停止します。
R_{Config_IT*m*0_IT*m*1}_Create_UserInit		IT*m* ユニットに関するユーザ独自の初期化処理を実行します。
r_{Config_IT*m*0_IT*m*1}_interrupt		16 ビット・インターバル・タイマ・モード割り込み（INTIT*m*0）に伴う処理を実行します。

R_{Config_ITm0_ITm1}_Create

16 ビット・インターバル・タイマ・モードで ITm モジュールを制御する前に必要な初期化処理を実行します。

備考 この API 関数は、R_ITm_Create から呼び出されます。

[指定形式]

```
void    R_{Config_ITm0_ITm1}_Create(void);
```

備考 m はユニット番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_IT*m*0_IT*m*1}_Start

IT*m* ユニットを起動します。

[指定形式]

```
void    R_{Config_ITm0_ITm1}_Start(void);
```

備考　　*m* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_IT*m*0_IT*m*1}_Stop

IT*m* ユニットを停止します。

[指定形式]

```
void    R_{Config_ITm0_ITm1}_Stop(void);
```

備考　　*m* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_IT*m*0_IT*m*1}_Create_UserInit

IT*m* ユニットに関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_IT*m*0_ITm1}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_ITm0_ITm1}_Create_UserInit(void);
```

備考　*m* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_IT*m*0_IT*m*1}_interrupt

16 ビット・インターバル・タイマ・モード割り込み（INTIT*m*0）に伴う処理を実行します。

備考　この API 関数は、カウント終了割り込み（INTIT*m*0）の割り込みハンドラとして呼び出され、カウンタ値（TRTCMP*m*）が 0000H に達したときに発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_IT*m*0_IT*m*1}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_IT*m*0_IT*m*1}_Callback_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_IT*m*0_IT*m*1}_interrupt(void);
```

備考　*m* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

使用例

ユーザ定義のカウンタ値に 16 ビット・カウントを使用し、P00 から波形を出力する例です。

main.c

```
#include "r_smc_entry.h"
extern volatile uint8_t interrupt_flag;

void main(void);

void main(void)
{
    EI();    // 割り込みを有効にします。
    interrupt_flag = 0;
    R_Config_IT00_IT01 _Start();    // IT00、IT01 動作を有効にします。
    while (interrupt_flag < 20);    // interrupt_flag が 20 以上になるまで待ち、ループを終了します。
    R_Config_IT00_IT01 _Stop();    // IT00、IT01 動作を無効にします。
}
```

Config_IT00_IT01_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t interrupt_flag;
/* End user code. Do not edit comment generated here */

void R_Config_IT00_IT01_Interrupt(void)
{
    /* Start user code for R_Config_IT01_Interrupt. Do not edit comment generated here */
    interrupt_flag ++;    // 割り込みハンドラが入力された回数をカウントします。
    P0 = ~P0;    // 割り込みが発生するたびに、P00 のレベルが 1 回反転します。
    /* End user code. Do not edit comment generated here */
}
```

4.2.27 インプットキャプチャ機能（タイマ RD）

以下に、スマート・コンフィグレータがインプットキャプチャ機能用として出力する API 関数の一覧を示します。

表 4.32 インプットキャプチャ機能（タイマ RD）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TRD*n*}_Create	タイマ RD	インプットキャプチャ機能モードで TRD*n* モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_TRD*n*}_Start		TRD*n* のカウンタを起動します。
R_{Config_TRD*n*}_Stop		TRD*n* のカウンタを停止します。
R_{Config_TRD*n*}_Get_PulseWidth		TRD*n* の入力パルス幅を測定します。
R_{Config_TRD*n*}_Create_UserInit		TRD*n* に関するユーザ独自の初期化処理を実行します。
r_{Config_TRDn}_trd*n*_interrupt		INTTRD*n* 割り込みに伴う処理を実行します。

R_{Config_TRD*n*}_Create

インプットキャプチャ機能モードで TRD*n* モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_TRD_Create から呼び出されます。

[指定形式]

```
void    R_{Config_TRDn}_Create(void);
```

備考　*n* = 0, 1

[引数]

なし

[戻り値]

なし

R_{Config_TRD*n*}_Start

TRD*n* のカウンタを起動します。

[指定形式]

```
void    R_{Config_TRDn}_Start(void);
```

備考　　*n* = 0, 1

[引数]

なし

[戻り値]

なし

R_{Config_TRD*n*}_Stop

TRD*n* のカウンタを停止します。

[指定形式]

```
void    R_{Config_TRDn}_Stop(void);
```

備考　　*n* = 0, 1

[引数]

なし

[戻り値]

なし

R_{Config_TRD*n*}_Get_PulseWidth

TRD*n* の入力パルス幅を測定します。

[指定形式]

```
MD_STATUS R_{Config_TRDn}_Get_PulseWidth (uint32_t * const active_width, uint32_t * const inactive_width, e_timer_channel_t channel);
```

備考　　*n* = 0, 1

[引数]

I/O	引数	説明
O	uint32_t * const active_width;	High レベル幅
O	uint32_t * const inactive_width;	Low レベル幅
I	e_timer_channel_t channel	TRDIO*ji* 端子（i = 0, 1, j = A, B, C, D）外部信号と ELC 信号入力。

備考　　以下に、構造体 e_timer_channel_t を示します。

```
typedef enum
{
    TMCHANNELA,
    TMCHANNELB,
    TMCHANNELC,
    TMCHANNELD,
    TMCHANNELELC
} e_timer_channel_t;
```

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_ERROR	カウンタがキャプチャ・モードとして機能しない
MD_ARGERROR	引数入力エラー

R_{Config_TRD*n*}_Create_UserInit

TRD*n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TRD*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TRDn}_Create_UserInit(void);
```

備考　*n* = 0, 1

[引数]

なし

[戻り値]

なし

r_{Config_TRD*n*}_trd*n*_interrupt

INTTRD*n* 割り込みに伴う処理を実行します。

備考　この API 関数は、キャプチャ割り込みの割り込みハンドラ（INTTRD*n*）として呼ばれ、TRDIO*jn*（*j* = A, B, C, D）入力の有効なキャプチャ・エッジが検出されたとき、またはTRD*n* レジスタがオーバーフローの場合に発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TRDn}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TRDn}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TRDn}_interrupt(void);
```

備考　*n* = 0, 1

[引数]

なし

[戻り値]

なし

使用例

TRD0 入力パルス幅を取得する例です。

main.c

```
#include "r_smc_entry.h"

void main(void);

typedef struct {
	uint32_t active_width;
	uint32_t inactive_width;
} TRD_PulseWidth_t;

TRD_PulseWidth_t trd_a;
TRD_PulseWidth_t trd_b;
TRD_PulseWidth_t trd_d;

static void delay_ms(uint32_t time_ms)
{
	uint32_t i = 0;
	while(time_ms--) {
		for(i = 0; i < 156; I++) {
			NOP();
		}
	}
}

void main(void)
{
	EI();	// 割り込みを有効にします。

	P0 = 0x00U;	// ポート 0 を使用して割り込みの頻度を確認します。
	R_Config_TAU0_0_Start();	// TAU00 を使用して TRD0 チャネル A のパルスを生成します。
	R_Config_TAU0_1_Start();	// TAU01 を使用して TRD0 チャネル B のパルスを生成します。
	R_Config_TAU0_2_Start();	// TAU02 を使用して TRD0 チャネル C のパルスを生成します。
	R_Config_TRD0_Start();	// TRD 動作を有効にします。

	delay_ms(2000);	// 開始から 2 秒待ちます。
	// 各チャンネルの active_width と inactive_width を読み取ります。
	R_Config_TRD0_Get_PulseWidth(&trd_a.active_width, &trd_a.inactive_width, TMCHANNELA);
	R_Config_TRD0_Get_PulseWidth(&trd_b.active_width, &trd_b.inactive_width, TMCHANNELB);
	R_Config_TRD0_Get_PulseWidth(&trd_d.active_width, &trd_d.inactive_width, TMCHANNELC);

	while(1);
}
```

Config_TRD0_user.c

```
static void __near r_Config_TRD0_interrupt(void)
{
    ...
    /* Start user code for r_Config_TRD0_interrupt. Do not edit comment generated here */

    /* TRDGRA0 input capture interrupt */
    P0_bit.no0 = ~P0_bit.no0;    // P00 を使用して割り込みの頻度を確認します。

    /* TRDGRB0 input capture interrupt */
    P0_bit.no1 = ~P0_bit.no1;    // P01 を使用して割り込みの頻度を確認します。

    /* TRDGRD0 input capture interrupt */
    P0_bit.no2 = ~P0_bit.no2;    // P02 を使用して割り込みの頻度を確認します。
    /* End user code. Do not edit comment generated here */
}
```

4.2.28 インプットキャプチャ機能（タイマ RG）

以下に、スマート・コンフィグレータがインプットキャプチャ機能用として出力する API 関数の一覧を示します。

表 4.33 インプットキャプチャ機能（タイマ RG）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TRG}_Create	タイマ RG	インプットキャプチャ機能モードで TRG モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_TRG}_Start		TRG のカウンタを起動します。
R_{Config_TRG}_Stop		TRG のカウンタを停止します。
R_{Config_TRG}_Get_PulseWidth		TRG の入力パルス幅を測定します。
R_{Config_TRG}_Create_UserInit		TRG に関するユーザ独自の初期化処理を実行します。
r_{Config_TRG}_interrupt		INTTRG 割り込みに伴う処理を実行します。

R_{Config_TRG}_Create

インプットキャプチャ機能モードで TRG モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_TRG}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TRG}_Start

TRG のカウンタを起動します。

[指定形式]

```
void    R_{Config_TRG}_Start(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TRG}_Stop

TRG のカウンタを停止します。

[指定形式]

```
void    R_{Config_TRG}_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TRG}_Get_PulseWidth

TRG の入力パルス幅を測定します。

[指定形式]

```
MD_STATUS R_{Config_TRG}_Get_PulseWidth (uint32_t * const active_width, uint32_t * const inactive_width, e_timer_channel_t channel);
```

[引数]

I/O	引数	説明
O	uint32_t * const active_width;	High レベル幅
O	uint32_t * const inactive_width;	Low レベル幅
I	e_timer_channel_t channel	TRGIOA, TRGIOB 端子外部信号と ELC 信号入力。

備考　以下に、構造体 e_trg_channel_t を示します。

```
typedef enum
{
    TRG_CHANNELA,
    TRG_CHANNELB,
    TRG_CHANNELELC
} e_trg_channel_t;
```

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_ERROR	カウンタがキャプチャ・モードとして機能しない
MD_ARGERROR	引数入力エラー

R_{Config_TRG}_Create_UserInit

TRG に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TRG}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TRG}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_TRG}_interrupt

INTTRG 割り込みに伴う処理を実行します。

備考　この API 関数は、キャプチャ割り込みの割り込みハンドラ（INTTRG）として呼ばれ、TRDIOA, TRDIOB 入力の有効なキャプチャ・エッジが検出されたとき、または TRG レジスタがオーバーフローの場合に発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TRG}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TRG}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TRG}_interrupt(void);
```

[引数]

なし

[戻り値]

なし

使用例

TRG 入力パルス幅を取得する例です。

main.c

```
#include "r_smc_entry.h"

void main(void);

typedef struct {
	uint32_t active_width;
	uint32_t inactive_width;
} TRG_PulseWidth_t;

TRG_PulseWidth_t trg_a;
TRG_PulseWidth_t trg_b;

static void delay_ms(uint32_t time_ms)
{
	uint32_t i = 0;
	while(time_ms--) {
		for(i = 0; i < 156; I++) {
			NOP();
		}
	}
}

void main(void)
{
	EI();	// 割り込みを有効にします。

	P0 = 0x00U;	// ポート 0 を使用して割り込みの頻度を確認します。
	R_Config_TAU0_0_Start();	// TAU00 を使用して TRD0 チャネル A のパルスを生成します。
	R_Config_TAU0_1_Start();	// TAU01 を使用して TRD0 チャネル B のパルスを生成します。
	R_Config_TRG_Start();	// TRG 動作を有効にします。

	delay_ms(2000);	// 開始から 2 秒待ちます。
	// 各チャンネルの active_width と inactive_width を読み取ります。
	R_Config_TRG_Get_PulseWidth(&trg_a.active_width, &trg_a.inactive_width, TMCHANNELA);
	R_Config_TRG_Get_PulseWidth(&trg_b.active_width, &trg_b.inactive_width, TMCHANNELB);

	while(1);
}
```

Config_TRG_user.c

```
static void __near r_Config_TRG_interrupt(void)
{
    ...
    /* Start user code for r_Config_TRG_interrupt. Do not edit comment generated here */

    /* TRGGRA input capture interrupt */
    P0_bit.no0 = ~P0_bit.no0;

    /* TRGGRB input capture interrupt */
    P0_bit.no1 = ~P0_bit.no1;
    /* End user code. Do not edit comment generated here */
}
```

4.2.29 インプットキャプチャ機能（タイマ RX）

以下に、スマート・コンフィグレータがインプットキャプチャ機能用として出力する API 関数の一覧を示します。

表 4.34 インプットキャプチャ機能（タイマ RD）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TRX}_Create	タイマ RX	インプットキャプチャ機能モードで TRX モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_TRX}_Start		TRX のカウンタを起動します。
R_{Config_TRX}_Stop		TRX のカウンタを停止します。
R_{Config_TRX}_Get_BufferValue		TRX の入力パルス幅を測定します。
R_{Config_TRX}_Create_UserInit		TRX に関するユーザ独自の初期化処理を実行します。
r_{Config_TRX}_interrupt		INTTRX 割り込みに伴う処理を実行します。

R_{Config_TRX}_Create

インプットキャプチャ機能モードで TRX モジュールを制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_TRX}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TRX}_Start

TRX のカウンタを起動します。

[指定形式]

```
void    R_{Config_TRX}_Start(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TRX}_Stop

TRX のカウンタを停止します。

[指定形式]

```
void    R_{Config_TRX}_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TRX}_Get_BufferValue

TRX のバッファ値を取得します。

[指定形式]

```
void R_{Config_TRX}_Get_BufferValue(uint32_t * const value);
```

[引数]

I/O	引数	説明
O	uint32_t * value;	バッファ値

[戻り値]

なし

R_{Config_TRX}_Create_UserInit

TRX に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TRX}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TRX}_Create_UserInit(void);
```

備考　*n* = 0, 1

[引数]

なし

[戻り値]

なし

r_{Config_TRX}_interrupt

INTTRX 割り込みに伴う処理を実行します。

備考　この API 関数は、キャプチャ割り込みの割り込みハンドラ（INTTRX）として呼ばれ、コンパレータ割り込み信号が検出されたとき、または TRX レジスタがオーバーフローの場合に発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TRX}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TRX}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TRX}_interrupt(void);
```

[引数]

なし

[戻り値]

なし

使用例

TRX 入力パルス幅を取得する例です。

main.c

```
#include "r_smc_entry.h"

uint32_t trx_buffer[10] = {0};
volatile uint8_t comp_flag = 0U;

void main(void);

void main(void)
{
    EI();   // 割り込みを有効にします。
    R_Config_TRX_Start();   // TRX 動作を有効にします。
    R_Config_COMP2_Start();   // タイマ RX カウンタは、コンパレータ 2 からのトリガでリセットされます。

    // 最初の割り込みはダミー値で、trx_buffer に格納する必要はありません。
    while(comp_flag != 1U);
    comp_flag = 0U;

    // コンパレータ 2 から割り込みが発生した場合、キャプチャ値をバッファに転送します。
    for (char i = 0; i < 10; i++)
    {
        while(comp_flag != 1U);
        R_Config_TRX_Get_BufferValue(trx_buffer + i);   // 10 回キャプチャします。
        comp_flag = 0U;
    }

    while(1);
}
```

Config_COPM0_user.c

```
extern volatile uint8_t comp_flag;
static void __near r_Config_COMP2_interrupt(void)
{
    ...
     /* Start user code for r_Config_TRX_interrupt. Do not edit comment generated here */
     comp_flag = 1U;   // コンパート 2 からのタイマ RX で使用するための割り込み出力信号
     /* End user code. Do not edit comment generated here */
}
```

4.2.30 ワンショット・パルス出力

以下に、スマート・コンフィグレータがワンショット・パルス出力機能用として出力する API 関数の一覧を示します。

表 4.35 ワンショット・パルス出力用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TAU*m*_*n*}_Create	タイマ・アレイ・ユニット	TAU*m* チャンネル・モジュールをワンショット・パルス出力モードで制御する前に必要な初期化処理を実行します。
R_{Config_TAU*m*_*n*}_Start		TAU*m* チャネル *n* のカウンタを起動します。
R_{Config_TAU*m*_*n*}_Stop		TAU*m* チャネル *n* のカウンタを停止します。
R_{Config_TAU*m*_*n*}_Set_SoftwareTriggerOn		ソフトウェア・トリガを発生させます。
R_{Config_TAU*m*_*n*}_Get_PulseWidth		TAU*m* チャネル *n* の入力パルス幅を測定します。
R_{Config_TAU*m*_*n*}_Create_UserInit		TAU*m* チャネル *n* に関するユーザ独自の初期化処理を実行します。
r_{Config_TAU*m*_*n*}_channel*n*_interrupt		INTTM*mn* 割り込みに伴う処理を実行します。
r_{Config_TAU*m*_*n*}_channel*p*_interrupt		INTTM*mp* 割り込みに伴う処理を実行します。

R_{Config_TAU*m*_*n*}_Create

TAU*m* チャンネル・モジュールをワンショット・パルス出力モードで制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_TAU*m*_Create から呼び出されます。

[指定形式]

```
void    R_{Config_TAUm_n}_Create(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Start

TAU*m* チャネル *n* のカウンタを起動します。

[指定形式]

```
void    R_{Config_TAUm_n}_Start(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Stop

TAU*m* チャネル *n* のカウンタを停止します。

[指定形式]

```
void    R_{Config_TAUm_n}_Stop(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Set_SoftwareTriggerOn

ソフトウェア・トリガを発生させます。

[指定形式]

```
void    R_{Config_TAUm_n}_Set_SoftwareTriggerOn(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Get_PulseWidth

TAU*m* チャネル *n* の入力パルス幅を測定します。

[指定形式]

void R_{Config_TAU*m*_*n*}_Get_PulseWidth(uint32_t * const width);

備考 *m* はユニット番号、*n* はチャネル番号を示します。

[引数]

I/O	引数	説明
O	uint32_t * const width;	入力パルス幅を格納するアドレス

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Create_UserInit

TAU*m* のチャネル *n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TAU*m*_*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TAUm_n}_Create_UserInit(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_TAU*m*_*n*}_channel*n*_interrupt

INTTM*mn* 割り込みに伴う処理を実行します。

備考　この API 関数は、カウント終了割り込み（INTTM*mn*）の割り込みハンドラとして呼び出され、カウンタ値（TCR*mn*）が 0000H に達したときに発生します。

［指定形式］

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TAUm_n}_channeln_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TAUm_n}_channeln_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TAUm_n}_channeln_interrupt(void);
```

備考　*m* はユニット番号、*n* はマスタ・チャネル番号を示します。

［引数］

なし

［戻り値］

なし

r_{Config_TAU*m_n*}_channel*p*_interrupt

INTTM*mp* 割り込みサービス・ルーチンです。

備考　この API 関数は、カウント終了割り込み（INTTM*mp*）の割り込みハンドラとして呼び出され、カウンタ値（TCR*mp*）が 0000H に達したときに発生します。

［指定形式］

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TAUm_n}_channelp_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TAUm_n}_channelp_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TAUm_n}_channelp_interrupt(void);
```

備考 1.　*m* はユニット番号、*n* はマスタ・チャネル番号、*p* はスレーブ・チャネル番号を示します。

備考 2.　$n < p \leqq 7$

［引数］

なし

［戻り値］

なし

使用例

ソフトウェア・トリガにより、ワンショット・パルスを出力する TAU チャネル 0 の例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t tau_oneshot_count;

void main(void);

void main(void)
{
    EI();   // 割り込みを有効にします。
    tau_oneshot_count = 0;
    R_Config_TAU0_0_Start();   // TAU00 動作を有効にします。
    R_Config_TAU0_0_Set_SoftwareTriggerOn();   // ソフトウェアで TS00 を 1 に設定します。
    while (tau_ oneshot_count < 10);
    {
        // タイマ割り込み発生ごとにソフトウェアで TS00 を 1 に設定します。
        R_Config_TAU0_0_Set_SoftwareTriggerOn();
    }
    R_Config_TAU0_0_Stop();   // TAU00 動作を無効にします。
}
```

Config_TAU0_0_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t tau_oneshot_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_01_channel1_interrupt (void)
{
  /* Start user code for r_Config_TAU0_01_channel1_interrupt. Do not edit comment generated here */
    tau_oneshot_count ++;   // 割り込みハンドラが入力された回数をカウントします。
  /* End user code. Do not edit comment generated here */
}
```

4.2.31 方形波出力（タイマ・アレイ・ユニット）

以下に、スマート・コンフィグレータが方形波出力機能用として出力する API 関数の一覧を示します。

表 4.36 方形波出力（タイマ・アレイ・ユニット）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TAU*m*_*n*}_Create	タイマ・アレイ・ユニット	方形波出力モードで TAU*m* チャネル・モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_TAU*m*_*n*}_Start		TAU*m* のチャネル *n* のカウンタを起動します。
R_{Config_TAU*m*_*n*}_Stop		TAU*m* のチャネル *n* のカウンタを停止します。
R_{Config_TAU*m*_*n*}_Lower8bits_Start		TAU*m* のチャネル *n* の下位側 8 ビット・カウンタを起動します。
R_{Config_TAU*m*_*n*}_Lower8bits_Stop		TAU*m* のチャネル *n* の下位側 8 ビット・カウンタを停止します。
R_{Config_TAU*m*_*n*}_Create_UserInit		TAU*m* のチャネル *n* に関するユーザ独自の初期化処理を実行します。
r_{Config_TAU*m*_*n*}_interrupt		INTTM*mn* 割り込みに伴う処理を実行します。

R_{Config_TAU*m*_*n*}_Create

方形波出力モードで TAU*m* チャネル・モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_TAU*m*_Create から呼び出されます。

[指定形式]

```
void    R_{Config_TAUm_n}_Create(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Start

TAU*m* のチャネル *n* のカウンタを起動します。

[指定形式]

```
void    R_{Config_TAUm_n}_Start(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Stop

TAU*m* のチャネル *n* のカウンタを停止します。

[指定形式]

```
void    R_{Config_TAUm_n}_Stop(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Lower8bits_Start

TAU*m* のチャネル *n* の下位側 8 ビット・カウンタを起動します。

[指定形式]

```
void    R_{Config_TAUm_n}_Lower8bits_Start(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号 1 または 3 を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Lower8bits_Stop

TAU*m* のチャネル *n* の下位側 8 ビット・カウンタを停止します。

[指定形式]

```
void    R_{Config_TAUm_n}_Lower8bits_Stop(void);
```

備考　　*m* はユニット番号、*n* はチャネル番号 1 または 3 を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TAU*m*_*n*}_Create_UserInit

TAU*m* のチャネル n に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TAU*m*_*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TAUm_n}_Create_UserInit(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_TAU*m*_*n*}_interrupt

INTTM*mn* 割り込みに伴う処理を実行します。

備考　この API 関数は、カウント終了割り込み（INTTM*mn*）の割り込みハンドラとして呼び出され、カウンタ値（TCR*mn*）が 0000H に達したときに発生します。

［指定形式］

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TAUm_n}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TAUm_n}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TAUm_n}_interrupt(void);
```

備考　*m* はユニット番号、*n* はチャネル番号を示します。

［引数］

なし

［戻り値］

なし

使用例

TAU チャネル 0 カウンタとチャネル 1 の下位 8 ビットカウンターを使用して、方形波出力する例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t ch0_run_count;
extern uint8_t ch1_run_count;

void main(void);

void main(void)
{
    EI();    // 割り込みを有効にします。
    R_Config_TAU0_0_Start();    // TAU00 動作を有効にします。
    while (ch0_run_count < 20);    // ch0_run_count が 20 以上になるまで待ち、ループを終了します。
    R_Config_TAU0_0_Stop();    // TAU00 動作を無効にします。

    R_Config_TAU0_1_Lower8bits_Start();    // TAU01 動作を有効にします。
    while (ch1_run_count < 20);    // ch1_run_count が 20 以上になるまで待ち、ループを終了します。
    R_Config_TAU0_1_Lower8bits_Stop();    // TAU01 動作を無効にします。

}
```

Config_TAU0_0_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch0_run_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_0_interrupt (void)
{
    /* Start user code for r_Config_ TAU0_0_interrupt. Do not edit comment generated here */
    ch0_run_count ++;    // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

Config_TAU0_1_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch1_run_count ;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TAU0_1_interrupt (void)
{
    /* Start user code for r_Config_TAU0_1_interrupt. Do not edit comment generated here */
    ch1_run_count ++;    // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

4.2.32 方形波出力（タイマ RJ）

以下に、スマート・コンフィグレータが方形波出力機能（タイマ RJ*n*）用として出力する API 関数の一覧を示します。

表 4.37 方形波出力（タイマ RJ*n*）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TRJ*n*}_Create	タイマ RJ*n*	方形波出力モードで TRJ*n* モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_TRJ*n*}_Start		TRJ*n* のカウンタを起動します。
R_{Config_TRJ*n*}_Stop		TRJ*n* のカウンタを停止します。
R_{Config_TRJ*n*}_Create_UserInit		TRJ*n* に関するユーザ独自の初期化処理を実行します。
r_{Config_TRJ*n*}_interrupt		INTTRJ*n* 割り込みに伴う処理を実行します。

R_{Config_TRJ*n*}_Create

方形波出力モードで TRJ*n* モジュールを制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_TRJn}_Create(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TRJ*n*}_Start

TRJ*n* のカウンタを起動します。

[指定形式]

```
void    R_{Config_TRJn}_Start(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TRJ*n*}_Stop

TRJ*n* のカウンタを停止します。

[指定形式]

```
void    R_{Config_TRJn}_Stop(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_TRJ*n*}_Create_UserInit

TRJ*n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TRJ*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TRJn}_Create_UserInit(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_TRJ*n*}_interrupt

INTTRJ*n* 割り込みに伴う処理を実行します。

備考　この API 関数は、TRJ*n* 割り込み（INTTRJ*n*）の割り込みハンドラとして呼び出され、カウント値が 0000H に達し、次のカウント・ソースが入力されたときに発生し、カウンタがアンダフローになります。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TRJn}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TRJn}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TRJn}_interrupt(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

使用例

TRJ0 カウントを使用して、ユーザ定義周期の JIO0 端子および JO0 端子から反転パルスを出力する例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t ch0_run_count;

void main(void);

void main(void)
{
    EI();    // 割り込みを有効にします。
    R_Config_TRJ0_Start();    // TRJ0 動作を有効にします。
    while (ch0_run_count < 20);    // ch0_run_count が 20 以上になるまで待ち、ループを終了します。
    R_Config_TRJ0_Stop();    // TRJ0 動作を無効にします。
}
```

Config_TRJ0_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch0_run_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TRJ0_interrupt (void)
{
    /* Start user code for r_Config_ TRJ0_interrupt. Do not edit comment generated here */
    ch0_run_count ++;    // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

4.2.33 アウトプットコンペア機能（タイマ RD）

以下に、スマート・コンフィグレータがアウトプットコンペア機能用として出力する API 関数の一覧を示します。

表 4.38 アウトプットコンペア機能（タイマ RD）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TRD*n*}_Create	タイマ RD	アウトプットコンペア機能モードで TRD*n* モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_TRD*n*}_Start		TRD*n* のカウンタを起動します。
R_{Config_TRD*n*}_Stop		TRD*n* のカウンタを停止します。
R_{Config_TRD*n*}_Create_UserInit		TRD*n* に関するユーザ独自の初期化処理を実行します。
r_{Config_TRDn}_trd*n*_interrupt		INTTRD*n* 割り込みに伴う処理を実行します。

R_{Config_TRD*n*}_Create

アウトプットコンペア機能モードで TRD*n* モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_TRD_Create から呼び出されます。

[指定形式]

```
void    R_{Config_TRDn}_Create(void);
```

備考　*n* = 0, 1

[引数]

なし

[戻り値]

なし

R_{Config_TRD*n*}_Start

TRD*n* のカウンタを起動します。

[指定形式]

```
void    R_{Config_TRDn}_Start(void);
```

備考　　*n* = 0, 1

[引数]

なし

[戻り値]

なし

R_{Config_TRD*n*}_Stop

TRD*n* のカウンタを停止します。

[指定形式]

```
void    R_{Config_TRDn}_Stop(void);
```

備考　　*n* = 0, 1

[引数]

なし

[戻り値]

なし

R_{Config_TRD*n*}_Create_UserInit

TRD*n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TRD*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TRDn}_Create_UserInit(void);
```

備考　*n* = 0, 1

[引数]

なし

[戻り値]

なし

r_{Config_TRD*n*}_trd*n*_interrupt

INTTRD*n* 割り込みに伴う処理を実行します。

備考　この API 関数は、カウント・コンペアー致割り込みの割り込みハンドラ（INTTRD*n*）として呼ばれ、TRD*n* レジスタの内容が TRDGR*jn*（*j* = A, B, C, D）レジスタの内容と一致するか、TRD*n* レジスタがオーバーフローの場合に発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TRDn}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TRDn}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TRDn}_interrupt(void);
```

備考　*n* = 0, 1

[引数]

なし

[戻り値]

なし

使用例

TRD1 を使用して TRDIO*j*1 端子から任意のレベルを出力する例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t ch1_run_count;

void main(void);

void main(void)
{
    EI();    // 割り込みを有効にします。
    R_Config_TRD1_Start();    // TRD1 動作を有効にします。
    while (ch1_run_count < 20);    // ch1_run_count が 20 以上になるまで待ち、ループを終了します。
    R_Config_TRD1_Stop();    // TRD1 動作を無効にします。
}
```

Config_TRD1_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch1_run_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TRD1_trd1_interrupt (void)
{
    /* Start user code for r_Config_ TRD1_trd1_interrupt. Do not edit comment generated here */
    ch1_run_count ++;    // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

4.2.34 アウトプットコンペア機能（タイマ RG）

以下に、スマート・コンフィグレータがアウトプットコンペア機能用として出力する API 関数の一覧を示します。

表 4.39 アウトプットコンペア機能（タイマ RG）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TRG}_Create	タイマ RG	アウトプットコンペア機能モードで TRG モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_TRG}_Start		TRG のカウンタを起動します。
R_{Config_TRG}_Stop		TRG のカウンタを停止します。
R_{Config_TRG}_Create_UserInit		TRG に関するユーザ独自の初期化処理を実行します。
r_{Config_TRG}_interrupt		INTTRG 割り込みに伴う処理を実行します。

R_{Config_TRG}_Create

アウトプットコンペア機能モードで TRG モジュールを制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_TRG}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TRG}_Start

TRG のカウンタを起動します。

[指定形式]

```
void    R_{Config_TRG}_Start(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TRG}_Stop

TRG のカウンタを停止します。

[指定形式]

```
void    R_{Config_TRG}_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TRG}_Create_UserInit

TRG に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TRG}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TRG}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_TRG}_interrupt

INTTRG 割り込みに伴う処理を実行します。

備考　この API 関数は、カウント・コンペアー致割り込みの割り込みハンドラ（INTTRG）として呼ばれ、TRG レジスタの内容が TRGGR*j*（*j* = A, B, C, D）レジスタの内容と一致するか、TRG レジスタがオーバーフローの場合に発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TRG}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TRG}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TRG}_interrupt(void);
```

[引数]

なし

[戻り値]

なし

使用例

TRG を使用して TRGIOj 端子から任意のレベルを出力する例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t ch1_run_count;

void main(void);

void main(void)
{
    EI();    // 割り込みを有効にします。
    R_Config_TRG_Start();    // TRG 動作を有効にします。
    while (ch1_run_count < 20);    // ch1_run_count が 20 以上になるまで待ち、ループを終了します。
    R_Config_TRG_Stop();    // TRG 動作を無効にします。
}
```

Config_TRG_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t ch1_run_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TRG_interrupt (void)
{
    /* Start user code for r_Config_ TRG_interrupt. Do not edit comment generated here */
    ch1_run_count ++;    // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

4.2.35 三相 PWM 出力（タイマ RD）

以下に、スマート・コンフィグレータが三相 PWM 出力（リセット同期 PWM モード／相補 PWM モード／拡張相補 PWM モードを使用するタイマ RD）用として出力する API 関数の一覧を示します。

表 4.40 三相 PWM 出力（セット同期 PWM モード／相補 PWM モード／拡張相補 PWM モードを使用するタイマ RD）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TRD0_TRD1}_Create	タイマ RD	リセット同期 PWM モード／相補 PWM モード／拡張相補 PWM モードで TRD0 および TRD2 モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_TRD0_TRD1}_Start		TRD0 および TRD1 のカウンタを起動します。
R_{Config_TRD0_TRD1}_Stop		TRD0 および TRD1 のカウンタを停止します。
R_{Config_TRD0_TRD1}_Set_TRD_ReloadTrigger		TRD0 および TRD1 バッファレジスタのリロードトリガを生成します。
R_{Config_TRD0_TRD1}_Create_UserInit		TRD0_TRD1 に関するユーザ独自の初期化処理を実行します。
r_{Config_TRD0_TRD1}_trd0_interrupt		INTTRD0 割り込みに伴う処理を実行します。
r_{Config_TRD0_TRD1}_trd1_interrupt		INTTRD1 割り込みに伴う処理を実行します。

R_{Config_TRD0_TRD1}_Create

リセット同期 PWM モード／相補 PWM モード／拡張相補 PWM モードで TRD0 および TRD2 モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_TRD_Create から呼び出されます。

[指定形式]

```
void    R_{Config_TRD0_TRD1}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TRD0_TRD1}_Start

リセット同期 PWM モードで TRD0 カウンタを起動するか、相補 PWM モード／拡張相補 PWM モードモードで TRD0 および TRD1 カウンタを起動します。

[指定形式]

```
void    R_{Config_TRD0_TRD1}_Start(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TRD0_TRD1}_Stop

リセット同期 PWM モードで TRD0 カウンタを停止するか、相補 PWM モード／拡張相補 PWM モードで TRD0 および TRD1 カウンタを停止します。

[指定形式]

```
void    R_{Config_TRD0_TRD1}_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TRD0_TRD1}_Set_TRD_ReloadTrigger

TRD0 および TRD1 バッファレジスタのリロードトリガを生成します。

[指定形式]

```
MD_STATUS  R_{Config_TRD0_TRD1}_Set_TRDn_ReloadTrigger  (st_extpwm_buffer_registers_t
* buffer);
```

[引数]

I/O	引数	説明
I	st_extpwm_buffer_registers_t * buffer;	バッファレジスタ値

備考　　構造体 st_extpwm_buffer_registers_t を以下に示します。

```
typedef struct {
    uint16_t trdgrd0;
    uint16_t trdcmpd0;
    uint16_t trdgrc1;
    uint16_t trdcmpc1;
    uint16_t trdgrd1;
    uint16_t trdcmpd1;
    uint16_t trdadtb0;
    uint16_t trdadtb1;
} st_extcompwm_buffer_registers_t;
```

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_ERROR	リロードトリガステータス待ち

R_{Config_TRD0_TRD1}_Create_UserInit

TRD0_TRD1 に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TRD0_TRD1}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TRD0_TRD1}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_TRD0_TRD1}_trd0_interrupt

リセット同期 PWM モード／相補 PWM モードのタイマ TRD0 カウント・コンペアー致割り込み（INTTRD0）に応答して、または拡張相補 PWM モードのデシメーション制御で割り込み要求信号 0（INTTRD0）に応答して処理を実行します。

備考 1　リセット同期 PWM モードの場合、カウント・コンペアー致割り込み（INTTRD0）の割り込みハンドラとして呼び出され、TRD0 レジスタの内容が TRDGR*j*0（*j* = A、B、C、D）レジスタの内容と一致するか、TRD0 レジスタがオーバーフローの場合に発生します。

備考 2　相補 PWM モードの場合、カウント・コンペアー致割り込み（INTTRD0）の割り込みハンドラとして呼び出され、TRD0 レジスタの内容が TRDGR*j*0（*j* = A、B、C、D）レジスタの内容と一致する場合に発生します。

備考 3　拡張相補 PWM モードの場合、TRD1 レジスタがオーバーフローの場合に発生する割り込み要求信号 0（INTTRD0）の割り込みハンドラとして呼び出されます。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TRD0_TRD1}_trd0_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TRD0_TRD1}_ trd0_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TRD0_TRD1}_ trd0_interrupt(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_TRD0_TRD1}_trd1_interrupt

拡張相補 PWM モードでデシメーション制御を使用して、タイマ TRD1 カウント・コンペアー致割り込み（INTTRD1）またはタイマ TRD1 割り込み要求信号 1（TRD1）に応答して処理を実行します。

備考 1 リセット同期 PWM モードの場合、カウント・コンペアー致割り込み（INTTRD1）の割り込みハンドラとして呼び出され、TRD1 レジスタの内容が TRDGRA1 および TRDGRB1 レジスタの内容と一致する場合に発生します。

備考 2 相補 PWM モードの場合、カウント・コンペアー致割り込み（INTTRD1）の割り込みハンドラとして呼び出され、TRD1 レジスタの内容が TRDGR*j*0（*j* = A, B, C, D）レジスタの内容と一致するか、TRD1 レジスタがアンダフローの場合に発生します。

備考 3 拡張相補 PWM モードの場合、TRD1 レジスタがアンダフローの場合に発生する割り込み要求信号 1（INTTRD1）の割り込みハンドラとして呼び出されます。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TRD0_TRD1}_trd1_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TRD0_TRD1}_ trd1_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TRD0_TRD1}_ trd1_interrupt(void);
```

[引数]

なし

[戻り値]

なし

使用例

拡張相補 PWM モードで同じ周期の対称または非対称 PWM 波形の 3 つの順相と 3 つの逆相を出力する例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t trd_pwm_count;

void main(void);

void main(void)
{
    EI();   // 割り込みを有効にします。
    R_Config_TRD0_TRD1_Start();   // TRD0 および TRD1 動作を有効にします。
    while (trd_pwm_count < 20);    // trd_pwm_count が 20 以上になるまで待ち、ループを終了します。
    R_Config_TRD0_TRD1_Stop();   // TRD0 および TRD1 動作を無効にします。
}
```

Config_TRD0_TRD1_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t trd_pwm_count;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TRD0_TRD1_trd0_interrupt (void)
{
    /* Start user code for r_Config_TRD0_trd0_interrupt. Do not edit comment generated here */
    trd_pwm_count ++;   // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}

static void __near r_Config_TRD0_TRD1_trd1_interrupt (void)
{
    /* Start user code for r_Config_TRD0_trd1_interrupt. Do not edit comment generated here */
    trd_pwm_count ++;   // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

4.2.36 PWM オプション・ユニット A（タイマ RD）

以下に、スマート・コンフィグレータが PWM オプション・ユニット A として出力する API 関数の一覧を示します。

表 4.41 PWM オプション・ユニット A（タイマ RD）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_PWMOPA}_Create	タイマ RD	PWM オプション・ユニット A を制御する前に必要な初期化処理を実行します。
R_{Config_PWMOPA}_Software_Release		ソフトウェアにより出力をリリースします。
R_{Config_PWMOPA}_Create_UserInit		PWM オプション・ユニット A に関するユーザ独自の初期化処理を実行します。

R_{Config_PWMOPA}_Create

PWM オプション・ユニット A を制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、R_TRD_Create から呼び出されます。

[指定形式]

```
void    R_{Config_PWMOPA}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_PWMOPA}_Software_Release

ソフトウェアにより出力をリリースします。

[指定形式]

```
void    R_{Config_PWMOPA}_Software_Release(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_PWMOPA}_Create_UserInit

PWM オプション・ユニット A に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_PWMOPA}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_PWMOPA}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

使用例

タイマ RD 出力端子 TRDIO*ji*（*j* = A, B, C, D; *i* = 0, 1）からのパルス出力の例です。ソフトウェア・トリガによって強制遮断を解除し、パルス出力を再開します。

main.c

```
#include "r_smc_entry.h"

void main(void);

void main(void)
{
    EI();    // 割り込みを有効にします。
    R_Config_PWMOPA_Software_Release();    // 強制的に遮断された出力を解放します。
    while (1);    // ここで停止します。
}
```

4.2.37 位相計数モード

以下に、スマート・コンフィグレータが位相計数モード機能用として出力する API 関数の一覧を示します。位相計数モードは、2 本の TRGCLKA, TRGCLKB 端子からの外部入力信号の位相差を検出し、TRG カウンタをアップ／ダウンカウントします。

表 4.42 API 関数

API 関数名	周辺機能	機能概要
R_{Config_TRG}_Create	タイマ RG	インプットキャプチャ機能モードで TRG モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_TRG}_Start		TRG のカウンタを起動します。
R_{Config_TRG}_Stop		TRG のカウンタを停止します。
R_{Config_TRG}_Get_MeasurementCapture		TRG 測定キャプチャ値を取得して、TRGCLKA および TRGCLKB の位相変化時間を計算します。
R_{Config_TRG}_Create_UserInit¥		TRG に関するユーザ独自の初期化処理を実行します。
r_{Config_TRG}_interrupt		INTTRG 割り込みに伴う処理を実行します。
r_{Config_TRG}_clear_interrupt		カウントコンペアマッチカウンタクリアおよび Z 信号検出カウンタクリア割り込み (INTGCR) に応じた処理を実行します。
r_{Config_TRG}_capture_interrupt		TRG TRGPMC カウントコンペアマッチ割り込み (INTPMC)に伴う処理を実行します。

R_{Config_TRG}_Create

インプットキャプチャ機能モードで TRG モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_TRG}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TRG}_Start

TRG のカウンタを起動します。

[指定形式]

```
void    R_{Config_TRG}_Start(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TRG}_Stop

TRG のカウンタを停止します。

[指定形式]

```
void    R_{Config_TRG}_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_TRG}_Get_MeasurementCapture

TRG 測定キャプチャ値を取得して、TRGCLKA および TRGCLKB の位相変化時間を計算します。

[指定形式]

```
void R_{Config_TRG}_Get_MeasurementCapture(uint16_t * const capture_value);
```

[引数]

I/O	引数	説明
O	uint16_t * const capture_value;	測定キャプチャ値

[戻り値]

なし

R_{Config_TRG}_Create_UserInit

TRG に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_TRG}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_TRG}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_TRG}_interrupt

INTTRG 割り込みに伴う処理を実行します。

備考　この API 関数は、カウントコンペアマッチ割り込みの割り込みハンドラ（INTTRG）として呼ばれ、TRG レジスタの内容と TRGGR*h* (*h* = A、B、C、D) レジスタの内容が一致した場合、または TRG レジスタがオーバーフローの場合に発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TRG}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TRG}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TRG}_interrupt(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_TRG}_clear_interrupt

TRG カウントコンペアマッチカウンタクリアおよび Z 信号検出カウンタクリア割り込み (INTGCR) に応じた処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TRG}_clear_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TRG}_clear_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TRG}_clear_interrupt(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_TRG}_capture_interrupt

TRG TRGPMC カウントコンペアマッチ割り込み (INTPMC)に伴う処理を実行します。

［指定形式］

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_TRG}_capture_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_TRG}_capture_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_TRG}_capture_interrupt(void);
```

［引数］

なし

［戻り値］

なし

使用例

TRGGRA コンペアマッチによる位相計数モードのクリアの例です。

main.c

```
#include "r_cg_macrodriver.h"
#include "Config_TAU0_0.h"
#include "Config_TAU0_3.h"
#include "Config_TRG.h"

extern uint8_t count;

void main(void);

void main(void)
{
    EI();

    // TRGCLKA への外部信号入力をサポートします。
    R_Config_TAU0_0_Start();
    // TRGCLKB への外部信号入力をサポートします。
    R_Config_TAU0_3_Start();
    R_Config_TRG_Start();    // TRG 動作を有効にします。

    while (count > 0U);    // count が 0 を超えるとループを終了し、割り込みハンドラが引き起こされたことを意味します。
    R_Config_TRG_Stop();    // TRG 動作を無効にします。
}
```

Config_TRG_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t count = 0U;
/* End user code. Do not edit comment generated here */

static void __near r_Config_TRG_interrupt (void)
{
    /* Start user code for r_Config_TRG_interrupt. Do not edit comment generated here */
    count ++;    // 割り込みハンドラが入力された回数をカウントします。
    /* End user code. Do not edit comment generated here */
}
```

4.2.38 クロック出力／ブザー出力制御回路

以下に、スマート・コンフィグレータがクロック出力／ブザー出力制御回路用として出力する API 関数の一覧を示します。

表 4.43 クロック出力／ブザー出力制御回路用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_PCLBUZ*n*}_Create	クロック出力／ブザー出力制御回路	PCLBUZ*n* モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_PCLBUZ*n*}_Start		PCLBUZ*n* モジュールを起動します。
R_{Config_PCLBUZ*n*}_Stop		PCLBUZ*n* モジュールを停止します。
R_{Config_PCLBUZ*n*}_Create_UserInit		PCLBUZ に関するユーザ独自の初期化処理を実行します。

R_{Config_PCLBUZ*n*}_Create

PCLBUZ*n* モジュールを制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_PCLBUZn}_Create(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_PCLBUZ*n*}_Start

PCLBUZn を起動します。

[指定形式]

```
void    R_{Config_PCLBUZn}_Start(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_PCLBUZ*n*}_Stop

PCLBUZn を停止します。

[指定形式]

```
void    R_{Config_PCLBUZn}_Stop(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_PCLBUZ*n*}_Create_UserInit

PCLBUZ に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_PCLBUZ*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_PCLBUZn}_Create_UserInit(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

使用例

PCLBUZ0 を使用する例です。

main.c

```
#include "r_smc_entry.h"

void main(void);

void main(void)
{
    EI();    // 割り込みを有効にします。
    R_Config_PCLBUZ0_Start();    // PCLBUZ0 動作を有効にします。
}
```

4.2.39 リアルタイム・クロック

以下に、スマート・コンフィグレータがリアルタイム・クロック用として出力する API 関数の一覧を示します。

表 4.44 リアルタイム・クロック用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_RTC}_Create	リアルタイム・クロック	リアルタイム・クロック・モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_RTC}_Start		リアルタイム・クロックを許可します。
R_{Config_RTC}_Stop		リアルタイム・クロックを禁止します。
R_{Config_RTC}_Set_HourSystem		クロックの時間制（12 時間制、24 時間制）を選択します。
R_{Config_RTC}_Set_CounterValue		リアルタイム・クロック値を変更します。
R_{Config_RTC}_Get_CounterValue		現在のリアルタイム・クロックの値を読み出し、変数に格納します。
R_{Config_RTC}_Set_ConstPeriodInterruptOn		定周期割り込みを許可します。
R_{Config_RTC}_Set_ConstPeriodInterruptOff		定周期割り込みを禁止します。
R_{Config_RTC}_Set_AlarmOn		アラーム動作を開始します。
R_{Config_RTC}_Set_AlarmOff		アラーム動作を終了します。
R_{Config_RTC}_Set_AlarmValue		アラーム値を設定します。
R_{Config_RTC}_Get_AlarmValue		アラーム値を取得します。
R_{Config_RTC}_Set_RTC1HZOn		RTC1HZ 出力を許可します。
R_{Config_RTC}_Set_RTC1HZOff		RTC1HZ 出力を禁止します。
R_{Config_RTC}_Create_UserInit		リアルタイム・クロックに関するユーザ独自の初期化処理を実行します。
r_{Config_RTC}_interrupt		INTRTC 割り込みに伴う処理を実行します。
r_{Config_RTC}_callback_constperiod		INTRTC 定周期割り込みに伴う処理を実行します。
r_{Config_RTC}_callback_alarm		INTRTC アラーム割り込みに伴う処理を実行します。

R_{Config_RTC}_Create

リアルタイム・クロック・モジュールを制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_RTC}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_RTC}_Start

リアルタイム・クロックを許可します。

[指定形式]

```
void    R_{Config_RTC}_Start(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_RTC}_Stop

リアルタイム・クロックを禁止します。

[指定形式]

```
void    R_{Config_RTC}_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_RTC}_Set_HourSystem

クロックの時間制（12 時間制、24 時間制）を選択します。

[指定形式]

```
MD_STATUS    R_{Config_RTC}_Set_HourSystem(e_rtc_hour_system_t hour_system);
```

[引数]

I/O	引数	説明
I	e_rtc_hour_system_t hour_system;	時間制の種類 HOUR12：12 時間制 HOUR24：24 時間制

備考　以下に、構造体 e_rtc_hour_system_t（時間制）を示します。

```
typedef enum
{
   HOUR12,
   HOUR24
 } e_rtc_hour_system_t;
```

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_BUSY1	ビジー1
MD_BUSY2	ビジー2
MD_ARGERROR	引数の指定が不正

R_{Config_RTC}_Set_CounterValue

リアルタイム・クロック値を変更します。

[指定形式]

```
MD_STATUS    R_{Config_RTC}_Set_CounterValue(st_rtc_counter_value_t counter_write_val);
```

[引数]

I/O	引数	説明
I	st_rtc_counter_value_t counter_write_val;	カウント値（BCD 値）

備考　以下に、構造体 st_rtc_counter_value_t（カウンタ条件）を示します。

```
typedef struct
{
   uint8_t sec;
   uint8_t min;
   uint8_t hour;
   uint8_t day;
   uint8_t week;
   uint8_t month;
   uint8_t year;
 } st_rtc_counter_value_t;
```

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_BUSY1	ビジー1
MD_BUSY2	ビジー2

R_{Config_RTC}_Get_CounterValue

現在のリアルタイム・クロックの値を読み出し、変数に格納します。

[指定形式]

```
MD_STATUS    R_{Config_RTC}_Get_CounterValue(st_rtc_counter_value_t * const
counter_read_val);
```

[引数]

I/O	引数	説明
I	st_rtc_counter_value_t * const counter_read_val;	現在のリアルタイム・クロックの値（BCD 値）

備考　構造体 st_rtc_counter_value_t については、R_{Config_RTC}_Set_CounterValue を参照してください。

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_BUSY1	ビジー1
MD_BUSY2	ビジー2

R_{Config_RTC}_Set_ConstPeriodInterruptOn

定周期割り込みを許可します。

[指定形式]

```
MD_STATUS    R_{Config_RTC}_Set_ConstPeriodInterruptOn(e_rtc_int_period_t period);
```

[引数]

I/O	引数	説明
I	e_rtc_int_period_t period;	INTRTC の定周期

備考　以下に、構造体 e_rtc_int_period_t period（周期条件）を示します。

```
typedef enum
{
   HALFSEC = 1U,
   ONESEC,
   ONEMIN,
   ONEHOUR,
   ONEDAY,
   ONEMONTH
 } e_rtc_int_period_t;
```

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_ARGERROR	引数の指定が不正

R_{Config_RTC}_Set_ConstPeriodInterruptOff

定周期割り込みを禁止します。

[指定形式]

```
void    R_{Config_RTC}_Set_ConstPeriodInterruptOff(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_RTC}_Set_AlarmOn

アラーム動作を開始します。

[指定形式]

```
void    R_{Config_RTC}_Set_AlarmOn(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_RTC}_Set_AlarmOff

アラーム動作を終了します。

[指定形式]

```
void    R_{Config_RTC}_Set_AlarmOff(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_RTC}_Set_AlarmValue

アラーム値を設定します。

[指定形式]

```
void    R_{Config_RTC}_Set_AlarmValue(st_rtc_alarm_value_t alarm_val);
```

[引数]

I/O	引数	説明
I	st_rtc_alarm_value_t alarm_val;	アラームの発生条件（曜日，時，分）（BCD 値）

備考　以下に、構造体 st_rtc_alarm_value_t alarm_val（アラーム条件）を示します。

```
typedef struct
{
   uint8_t alarmwm;
   uint8_t alarmwh;
   uint8_t alarmww;
} st_rtc_alarm_value_t;
```

[戻り値]

なし

R_{Config_RTC}_Get_AlarmValue

アラーム値を取得します。

[指定形式]

```
void    R_{Config_RTC}_Get_AlarmValue (st_rtc_alarm_value_t * const alarm_val);
```

[引数]

I/O	引数	説明
O	st_rtc_alarm_value_t * const alarm_val;	取得したアラーム値（BCD 値）を格納するアドレス

備考　構造体 st_rtc_alarm_value_t * const alarm_val については、R_{Config_RTC}_Set_AlarmValue を参照してください。

[戻り値]

なし

R_{Config_RTC}_Set_RTC1HZOn

RTC1HZ 出力を許可します。

[指定形式]

```
void    R_{Config_RTC}_Set_RTC1HZOn(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_RTC}_Set_RTC1HZOff

RTC1HZ 出力を禁止します。

[指定形式]

```
void    R_{Config_RTC}_Set_RTC1HZOff(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_RTC}_Create_UserInit

リアルタイム・クロックに関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_RTC}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_RTC}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_RTC}_interrupt

INTRTC 割り込みに伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_RTC}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_RTC}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_RTC}_interrupt(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_RTC}_callback_constperiod

INTRTC 定周期割り込みに伴う処理を実行します。

備考　この API 関数は、定周期割り込みに伴う処理 r_{Config_RTC}_interrupt のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
static void    r_{Config_RTC}_callback_constperiod(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_RTC}_callback_alarm

INTRTC アラーム割り込みに伴う処理を実行します。

備考　この API 関数は、アラーム割り込みに伴う処理 r_{Config_RTC}_interrupt のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
static void    r_{Config_RTC}_callback_alarm(void);
```

[引数]

なし

[戻り値]

なし

使用例 1（アラーム割り込み）

アラーム割り込みを使用して、うるう秒補正を実行する例です（スケジュールされた日の 23:59:59 から 23:59:58 に時計を戻す）

main.c

```
#include "r_smc_entry.h"

void main(void);

void main(void)
{
    EI();   // 割り込みを有効にします。
    R_Config_RTC_Set_AlarmOn();   // アラーム動作を開始します。
    R_ Config_RTC_Start();   // リアルタイム・クロック・カウンタ動作を有効にします。
}
```

Config_RTC_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile st_rtc_counter_value_t counter_val;
/* End user code. Do not edit comment generated here */

static void r_Config_RTC_callback_alarm(void)
{
    /* Start user code for r_Config_RTC_callback_alarm. Do not edit comment generated here */
    // リアルタイム・クロックの結果を読み取り、変数に格納します。
    R_Config_RTC_Get_CounterValue ((st_rtc_counter_value_t *)&counter_val);

    /* Change the seconds */
    counter_val.rseccnt = 0x58U;

    R_Config_RTC_Set_CounterValue (counter_val);   // リアルタイム・クロック・カウンタ値を変更します。
    /* End user code. Do not edit comment generated here */
}
```

使用例 2（定周期割り込み）

定周期割り込みを使用して、1 時間ごとに割り込みを生成するように実装する例です。

main.c

```
#include "r_smc_entry.h"
st_rtc_counter_value_t  currTime;
st_rtc_alarm_value_t alarm;

void main(void);

void main(void)
{
    EI();    // 割り込みを有効にします。
    R_Config_RTC_Set_ConstPeriodInterruptOn(ONEHOUR);    // 1 時間に 1 回生成される定常的な周期割り込み、割り込みハンドラを有効にします。
    R_ Config_RTC_Start();    // リアルタイム・クロック・カウンタ動作を有効にします。
}
```

Config_RTC_user.c

```
/* Start user code for global. Do not edit comment generated here */
st_rtc_counter_value_t currTime;
st_rtc_alarm_value_t alarm;
/* End user code. Do not edit comment generated here */

static void r_Config_RTC_callback_constperiod (void)
{
    /* Start user code for r_Config_RTC_callback_constperiod. Do not edit comment generated here */
    // リアルタイム・クロックの結果を読み取り、変数に格納します。
    R_Config_RTC_Get_CounterValue(&currTime);
    // リアルタイム・クロックのアラーム値を読み取り、変数に格納します。
    R_Config_RTC_Get_AlarmValue(&alarm);
    // アラーム値を現在のカウンタ値に変更します。
    alarm.alarmww = currTime.week;
    alarm.alarmwh = currTime.hour;
    alarm.alarmwm = currTime.min + 5;
    R_Config_RTC_Set_AlarmValue(alarm);    // アラーム値を設定します。
    R_Config_RTC_Set_AlarmOn();    // アラーム動作を開始する。
    /* End user code. Do not edit comment generated here */
}
```

4.2.40 A/D コンバータ

以下に、スマート・コンフィグレータが A/D コンバータ用として出力する API 関数の一覧を示します。

表 4.45 A/D コンバータ用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_ADC}_Create	A/D コンバータ	ADC モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_ADC}_Start		A/D コンバータを起動します。
R_{Config_ADC}_Stop		A/D コンバータを停止します。
R_{Config_ADC}_Set_OperationOn		A/D 電圧コンパレータの動作を許可します。
R_{Config_ADC}_Set_OperationOff		A/D 電圧コンパレータの動作を禁止します。
R_{Config_ADC}_Set_ADChannel		アナログ入力チャネルを選択します。
R_{Config_ADC}_ADSn_Set_ADChannel		アナログ入力チャンネルを選択します。（「AD アドバンスド・モード」選択時のみ）
R_{Config_ADC}_Set_SnoozeOn		A/D ウエイクアップ機能を許可します。
R_{Config_ADC}_Set_SnoozeOff		A/D ウエイクアップ機能を禁止します。
R_{Config_ADC}_Set_TestChannel		テスト機能を設定します。
R_{Config_ADC}_Get_Result_10bit		バッファ内の上位 10 ビットの変換結果を返します。
R_{Config_ADC}_Get_Result_8bit		バッファ内の上位 8 ビットの変換結果を返します。
R_{Config_ADC}_Get_Result_12bit		バッファ内の上位 12 ビットの変換結果を返します。
R_{Config_ADC}_ADSn_Get_Result_10bit		バッファ内の上位 10 ビットの変換結果を返します。（「AD アドバンスド・モード」選択時のみ）
R_{Config_ADC}_ADSn_Get_Result_8bit		バッファ内の上位 8 ビットの変換結果を返します。（「AD アドバンスド・モード」選択時のみ）
R_{Config_ADC}_ADSn_Get_Result_12bit		バッファ内の上位 12 ビットの変換結果を返します。（「AD アドバンスド・モード」選択時のみ）
R_{Config_ADC}_Create_UserInit		A/D コンバータに関するユーザ独自の初期化処理を実行します。
r_{Config_ADC}_interrupt		INTAD 割り込みに伴う処理を実行します。
r_{Config_ADC}_adn_interrupt		INTAD 割り込みに伴う処理を実行します。（「AD アドバンスド・モード」選択時のみ）

R_{Config_ADC}_Create

ADC モジュールを制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_ADC}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_ADC}_Start

A/D コンバータを起動します。

[指定形式]

```
void    R_{Config_ADC}_Start(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_ADC}_Stop

A/D コンバータを停止します。

[指定形式]

```
void    R_{Config_ADC}_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_ADC}_Set_OperationOn

A/D 電圧コンパレータの動作を許可します。

[指定形式]

```
void    R_{Config_ADC}_Set_OperationOn(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_ADC}_Set_OperationOff

A/D 電圧コンパレータの動作を禁止します。

[指定形式]

```
void    R_{Config_ADC}_Set_OperationOff(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_ADC}_Set_ADChannel

アナログ入力チャネルを選択します。

[指定形式]

```
MD_STATUS    R_{Config_ADC}_Set_ADChannel(e_ad_channel_t channel);
```

[引数]

I/O	引数	説明
I	e_ad_channel_t channel;	アナログ入力チャネル

備考　以下に、構造体 e_ad_channel_t channel（チャネル条件）を示します。

```
typedef enum
{
    ADCHANNEL0, ADCHANNEL1, ADCHANNEL2, ADCHANNEL3, ADCHANNEL4,
      ADCHANNEL5, ADCHANNEL6, ADCHANNEL7, ADCHANNEL8, ADCHANNEL9,
      ADCHANNEL10, ADCHANNEL11, ADCHANNEL12, ADCHANNEL13,
      ADCHANNEL14, ADCHANNEL16 = 16U, ADCHANNEL17, ADCHANNEL18,
      ADCHANNEL19, ADCHANNEL20, ADCHANNEL21, ADCHANNEL22,
      ADCHANNEL23, ADCHANNEL24, ADCHANNEL25, ADCHANNEL26,
      ADTEMPERSENSOR0 = 128U, ADINTERREFVOLT
} e_ad_channel_t;
```

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_ARGERROR	引数の指定が不正

R_{Config_ADC}_ADS*n*_Set_ADChannel

アナログ入力チャンネルを選択します。(「AD アドバンスド・モード」選択時のみ)

[指定形式]

```
MD_STATUS    R_{Config_ADC}_ADSn_Set_ADChannel(e_ad_channel_t channel);
```

[引数]

I/O	引数	説明
I	e_ad_channel_t channel;	アナログ入力チャネル

備考　以下に、構造体 e_ad_channel_t channel（チャネル条件）を示します。

```
typedef enum
{
   ADCHANNEL0, ADCHANNEL1, ADCHANNEL2, ADCHANNEL3, ADCHANNEL4,
      ADCHANNEL5, ADCHANNEL6, ADCHANNEL7, ADCHANNEL8, ADCHANNEL9,
      ADCHANNEL10, ADCHANNEL11, ADCHANNEL12, ADCHANNEL13,
      ADCHANNEL14, ADCHANNEL16 = 16U, ADCHANNEL17, ADCHANNEL18,
      ADCHANNEL19, ADCHANNEL20, ADCHANNEL21, ADCHANNEL22,
      ADCHANNEL23, ADCHANNEL24, ADCHANNEL25, ADCHANNEL26,
      ADTEMPERSENSOR0 = 128U, ADINTERREFVOLT
 } e_ad_channel_t;
```

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_ARGERROR	引数の指定が不正

R_{Config_ADC}_Set_SnoozeOn

A/D ウエイクアップ機能を許可します。

[指定形式]

```
void    R_{Config_ADC}_Set_SnoozeOn(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_ADC}_Set_SnoozeOff

A/D ウエイクアップ機能を禁止します。

[指定形式]

```
void    R_{Config_ADC}_Set_SnoozeOn(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_ADC}_Set_TestChannel

テスト機能を設定します。

[指定形式]

```
MD_STATUS    R_{Config_ADC}_Set_TestChannel(e_test_channel_t channel);
```

[引数]

I/O	引数	説明
I	e_test_channel_t channel;	テストチャネル設定

備考　以下に、構造体 e_test_channel_t channel（入力チャネル条件）を示します。

```
typedef enum
{
   ADNORMALINPUT,
   ADAVREFM = 2U,
   ADAVREFP
 } e_test_channel_t;
```

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_ARGERROR	引数の指定が不正

R_{Config_ADC}_Get_Result_10bit

バッファ内の上位 10 ビットの変換結果を返します。

［指定形式］

```
void    R_{Config_ADC}_Get_Result_10bit(uint16_t * const buffer);
```

［引数］

I/O	引数	説明
I	uint16_t * const buffer;	変換結果を格納するアドレス

［戻り値］

なし

R_{Config_ADC}_Get_Result_8bit

バッファ内の上位 8 ビットの変換結果を返します。

［指定形式］

```
void    R_{Config_ADC}_Get_Result_8bit(uint8_t * const buffer);
```

［引数］

I/O	引数	説明
I	uint8_t * const buffer;	変換結果を格納するアドレス

［戻り値］

なし

R_{Config_ADC}_Get_Result_12bit

バッファ内の上位 12 ビットの変換結果を返します。

［指定形式］

```
void    R_{Config_ADC}_Get_Result_12bit(uint16_t * const buffer);
```

［引数］

I/O	引数	説明
I	uint16_t * const buffer;	変換結果を格納するアドレス

［戻り値］

なし

R_{Config_ADC}_ADS*n*_Get_Result_10bit

バッファ内の上位 10 ビットの変換結果を返します。(「AD アドバンスド・モード」選択時のみ)

[指定形式]

```
void    R_{Config_ADC}_ADSn_Get_Result_10bit(uint16_t * const buffer);
```

[引数]

I/O	引数	説明
I	uint16_t * const buffer;	変換結果を格納するアドレス

[戻り値]

なし

R_{Config_ADC}_ADS*n*_Get_Result_8bit

バッファ内の上位 8 ビットの変換結果を返します。(「AD アドバンスド・モード」選択時のみ)

[指定形式]

```
void    R_{Config_ADC}_ADSn_Get_Result_8bit(uint8_t * const buffer);
```

[引数]

I/O	引数	説明
I	uint8_t * const buffer;	変換結果を格納するアドレス

[戻り値]

なし

R_{Config_ADC}_ADS*n*_Get_Result_12bit

バッファ内の上位 12 ビットの変換結果を返します。(「AD アドバンスド・モード」選択時のみ)

[指定形式]

```
void    R_{Config_ADC}_ADSn_Get_Result_12bit(uint16_t * const buffer);
```

[引数]

I/O	引数	説明
I	uint16_t * const buffer;	変換結果を格納するアドレス

[戻り値]

なし

R_{Config_ADC}_Create_UserInit

A/D コンバータに関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_ADC}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_ADC}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_ADC}_interrupt

INTAD 割り込みに伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_ADC}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_ADC}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_ADC}_interrupt(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_ADC}_ad*n*_interrupt

INTAD 割り込みに伴う処理を実行します。(「AD アドバンスド・モード」選択時のみ)

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_ADC}_ad n_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_ADC}_ad n_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_ADC}_ad n_interrupt(void);
```

[引数]

なし

[戻り値]

なし

使用例 1（ノーマル・モード）

ノーマル・モードの 8 ビット A/D 変換結果を取得する例です。

main.c

```
#include "r_smc_entry.h"
uint8_t  adc_data[1] = {0};    // 8 ビット A/D 変換結果を格納します。
extern uint8_t  adc_Interrupt_flag;

void main(void);

void main(void)
{
      EI();    // 割り込みを有効にします。
      adc_Interrupt_flag = 0U;
      /* Start comparator 0 */
      R_Config_ADC_Set_OperationOn();    // A/D コンパレータを起動します。
      R_Config_ADC_Start ();    // A/D 変換を開始します。
      while(adc_Interrupt_flag != 1U);    // A/D 変換が完了するまで待ちます。
      R_Config_ADC_Get_Result_8bit(adc_data);    // 8 ビット A/D 変換結果を取得し、adc_data 配列に
格納します。
      R_Config_ADC_Stop ();    // A/D 変換を停止します。
      R_Config_ADC_Set_OperationOff();    // A/D コンパレータを停止します。
}
```

Config_ADC_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t  adc_Interrupt_flag;
/* End user code. Do not edit comment generated here */

static void __near r_Config_COMP0_interrupt(void)
{
      /* Start user code for r_Config_COMP0_interrupt. Do not edit comment generated here */
      // A/D 変換が完了したことを示す割り込みフラグを設定します。
      adc_Interrupt_flag = 1U;
      /* End user code. Do not edit comment generated here */
}
```

使用例 2（アドバンスド・モード）

アドバンスド・モードの 10 ビット A/D 変換結果を取得する例です。

main.c

```
#include "r_smc_entry.h"
Uint16_t  adc_data[2] = {0};    // 2 つの 10 ビット A/D 変換結果を格納します。
extern uint8_t  adc_Interrupt_flag = 0U;

void main(void);

void main(void)
{
      EI();    // 割り込みを有効にします。
      // ANI4 端子をアナログ入力として構成します。
      PMCA2 |= 0x10U;
      PM2 |= 0x10U;

      R_Config_ADC_Set_OperationOn();    // A/D コンパレータを起動します。
      R_Config_ADC_Start();    // A/D 変換を開始します。

      R_Config_ADC_Set_SoftwareTriggerOn();    // ソフトウェアによる A/D 変換トリガです。
      while(adc_Interrupt_flag != 1);    // 割り込みフラグが設定されるまで待機（変換完了）します。
      R_Config_ADC_ADS0_Get_Result_10bit(adc_data);       // 最初の 10 ビット A / D 結果を取得し、
adc_data[0] に格納します。
      R_Config_ADC_ADS0_Set_ADChannel(ADCHANNEL4);    // A/C チャンネルを ADCHANNEL4
(ANI4) に設定します。

      for(char i=0; i<100; i++);
      adc_Interrupt_flag = 0U;
      R_Config_ADC_Set_SoftwareTriggerOn();      // ソフトウェアによる 2 回目の ADC 変換トリガです。
      while(adc_Interrupt_flag != 1);    // A/D 変換が完了するまで待ちます。
      R_Config_ADC_ADS0_Get_Result_10bit(adc_data + 1);      // 2 回目の結果を取得し、adc_data[1]
に格納します。

      R_Config_ADC_Stop ();    // A/D 変換を停止します。
      R_Config_ADC_Set_OperationOff();    // A/D コンパレータを停止します。
      while(1);
}
```

Config_ADC_user.c

```
/* Start user code for global. Do not edit comment generated here */
extern volatile uint8_t adc_Interrupt_flag;
/* End user code. Do not edit comment generated here */

static void __near r_Config_COMP0_interrupt(void)
{
      /* Start user code for r_Config_COMP0_interrupt. Do not edit comment generated here */
      // A/D 変換が完了したことを示す割り込みフラグを設定します。
      adc_Interrupt_flag = 1U;
      /* End user code. Do not edit comment generated here */
}
```

4.2.41 12 ビット A/D シングル・スキャン

以下に、スマート・コンフィグレータが 12 ビット A/D シングル・スキャン用として出力する API 関数の一覧を示します。

表 4.46 12 ビット A/D シングル・スキャン用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_S12AD*n*}_Create	12 ビット A/D コンバータ	12 ビット A/D コンバータ・モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_S12AD*n*}_Start		12 ビット A/D コンバータを起動します。
R_{Config_S12AD*n*}_Stop		12 ビット A/D コンバータを停止します。
R_{Config_S12AD*n*}_Get_Result_12bit		バッファに 12 ビットの変換結果を返します。
R_{Config_S12AD*n*}_Create_UserInit		12 ビット A/D コンバータに関するユーザ独自の初期化処理を実行します。
r_{Config_S12AD*n*}_interrupt		INTAD 割り込みに伴う処理を実行します。

R_{Config_S12AD*n*}_Create

12 ビット A/D コンバータ・モジュールを制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_S12AD n}_Create(void);
```

備考　　*n* = 0

[引数]

なし

[戻り値]

なし

R_{Config_S12AD*n*}_Start

12 ビット A/D コンバータを起動します。

[指定形式]

```
void    R_{Config_S12AD n}_Start(void);
```

備考　　*n* = 0

[引数]

なし

[戻り値]

なし

R_{Config_S12AD*n*}_Stop

12 ビット A/D コンバータを停止します。

[指定形式]

```
void    R_{Config_S12ADn}_Stop(void);
```

備考　　*n* = 0

[引数]

なし

[戻り値]

なし

R_{Config_S12AD*n*}_Get_ValueResult

バッファに 12 ビットの変換結果を返します。

[指定形式]

```
void    R_{Config_S12AD n}_Get_ValueResult (e_ad_channel_t channel, uint16_t * const buffer);
```

備考　　*n* = 0

[引数]

I/O	引数	説明
I	e_ad_channel_t channel;	読み取るデータレジスタのチャネル
I	uint16_t * const buffer;	変換結果を格納するアドレス

備考　　以下に、構造 e_ad_channel_t チャネル（チャネル条件）を示します。

```
typedef enum
{
ADCHANNEL0, ADCHANNEL1, ADCHANNEL2, ADCHANNEL3, ADCHANNEL4,
ADCHANNEL5, ADCHANNEL6, ADCHANNEL7, ADCHANNEL8, ADCHANNEL9,
ADCHANNEL10, ADCHANNEL11, ADCHANNEL12, ADCHANNEL13, ADCHANNEL14,
ADCHANNEL15, ADCHANNEL16, ADCHANNEL17, ADCHANNEL18, ADCHANNEL19,
ADCHANNEL20, ADCHANNEL21, ADCHANNEL22, ADCHANNEL23, ADCHANNEL24,
ADCHANNEL25, ADCHANNEL26, ADCHANNEL27, ADCHANNEL28, ADCHANNEL29,
ADCHANNEL30, ADINTERREFVOLT, ADSELFDIAGNOSIS
} e_ad_channel_t;
```

[戻り値]

なし

R_{Config_S12AD*n*}_Create_UserInit

12 ビット A/D コンバータに関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_S12AD*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_S12ADn}_Create_UserInit(void);
```

備考　*n* = 0

[引数]

なし

[戻り値]

なし

r_{Config_S12AD*n*}_interrupt

INTAD 割り込みに伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_S12AD n}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_S12AD n}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_S12AD n}_interrupt(void);
```

備考　　*n* = 0

[引数]

なし

[戻り値]

なし

使用例

シングルス・キャン・モードの 12 ビット A/D 変換結果を取得する例です。

main.c

```
#include "r_smc_entry.h"

void main(void);

extern volatile uint8_t interrupt_flag;
uint16_t AD_buffer_0 = 0;    // チャネル 0 からの A/D 結果を確認する変数です。
uint16_t AD_buffer_30 = 0;    // チャネル 30 からの A/D 結果を確認する変数です。

void main(void)
{
    EI();
    interrupt_flag = 0;
    R_Config_S12AD0_Start();    // シングルス・キャン・モードで A/D 変換を開始します。
    while(   interrupt_flag != 1 );     // A/D 変換が完了するまで待機します。(割り込みフラグが設定されています)
    R_Config_S12AD0_Get_ValueResult (ADCHANNEL0, &AD_buffer_0);    // チャネル 0 から 12 ビット A/D 結果を取得します。
    R_Config_S12AD0_Get_ValueResult (ADCHANNEL30, &AD_buffer_30);     // チャネル 30 から 12 ビット A/D 結果を取得します。
    interrupt_flag = 2;    // データが読み取られたことを示すフラグを設定します。

    while(1);
}
```

Config_ADC_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint8_t interrupt_flag;
/* End user code. Do not edit comment generated here */

static void __near r_Config_S12AD0_interrupt(void)
{
    /* Start user code for r_Config_S12AD0_interrupt. Do not edit comment generated here */
    interrupt_flag = 1;    // 割り込みフラグを設定して、A/D 変換が完了したことを示します。
    /* End user code. Do not edit comment generated here */
}
```

4.2.42 12 ビット A/D 連続スキャン

以下に、スマート・コンフィグレータが 12 ビット A/D 連続スキャン用として出力する API 関数の一覧を示します。

表 4.47 12 ビット A/D 連続スキャン用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_S12AD*n*}_Create	12 ビット A/D コンバータ	12 ビット A/D コンバータ・モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_S12AD*n*}_Start		12 ビット A/D コンバータを起動します。
R_{Config_S12AD*n*}_Stop		12 ビット A/D コンバータを停止します。
R_{Config_S12ADn}_Get_ValueResult		バッファに 12 ビットの変換結果を返します。
R_{Config_S12AD*n*}_Create_UserInit		12 ビット A/D コンバータに関するユーザ独自の初期化処理を実行します。
r_{Config_S12AD*n*}_interrupt		INTAD 割り込みに伴う処理を実行します。

R_{Config_S12AD*n*}_Create

12 ビット A/D コンバータ・モジュールを制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_S12AD n}_Create(void);
```

備考　　*n* = 0

[引数]

なし

[戻り値]

なし

R_{Config_S12AD*n*}_Start

12 ビット A/D コンバータを起動します。

[指定形式]

```
void    R_{Config_S12ADn}_Start(void);
```

備考　　*n* = 0

[引数]

なし

[戻り値]

なし

R_{Config_S12AD*n*}_Stop

12 ビット A/D コンバータを停止します。

[指定形式]

void R_{Config_S12AD*n*}_Stop(void);

備考 *n* = 0

[引数]

なし

[戻り値]

なし

R_{Config_S12AD*n*}_Get_ValueResult

バッファに 12 ビットの変換結果を返します。

[指定形式]

void R_{Config_S12AD*n*}_Get_ValueResult (e_ad_channel_t channel, uint16_t * const buffer);

備考 *n* = 0

[引数]

I/O	引数	説明
I	e_ad_channel_t channel;	読み取るデータレジスタのチャネル
I	uint16_t * const buffer;	変換結果を格納するアドレス

備考 以下に、構造 e_ad_channel_t チャネル（チャネル条件）を示します。

```
typedef enum
{
ADCHANNEL0, ADCHANNEL1, ADCHANNEL2, ADCHANNEL3, ADCHANNEL4,
ADCHANNEL5, ADCHANNEL6, ADCHANNEL7, ADCHANNEL8, ADCHANNEL9,
ADCHANNEL10, ADCHANNEL11, ADCHANNEL12, ADCHANNEL13, ADCHANNEL14,
ADCHANNEL15, ADCHANNEL16, ADCHANNEL17, ADCHANNEL18, ADCHANNEL19,
ADCHANNEL20, ADCHANNEL21, ADCHANNEL22, ADCHANNEL23, ADCHANNEL24,
ADCHANNEL25, ADCHANNEL26, ADCHANNEL27, ADCHANNEL28, ADCHANNEL29,
ADCHANNEL30, ADSELFDIAGNOSIS
} e_ad_channel_t;
```

[戻り値]

なし

R_{Config_S12AD*n*}_Create_UserInit

12 ビット A/D コンバータに関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_S12AD*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_S12ADn}_Create_UserInit(void);
```

備考　*n* = 0

[引数]

なし

[戻り値]

なし

r_{Config_S12AD*n*}_interrupt

INTAD 割り込みに伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_S12AD n}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_S12AD n}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_S12AD n}_interrupt(void);
```

備考　　*n* = 0

[引数]

なし

[戻り値]

なし

使用例

連続スキャン・モードの 12 ビット A/D 変換結果を TAU ソフトウェア・トリガ入力で取得する例です。

main.c

```
#include "r_smc_entry.h"

void main(void);

extern volatile uint8_t interrupt_flag;
uint16_t AD_buffer_3 = 0;    // チャネル 3 からの A/D 結果を格納するバッファです。
uint16_t AD_buffer_28 = 0;    // チャネル 28 からの A/D 結果を格納するバッファです。
uint8_t continues_num = 0;    // A/D 変換回数カウンタです。

void main(void)
{
      EI();

      interrupt_flag = 0;
      continues_num = 0;
      R_Config_S12AD0_Start();    // A/D 変換を開始します。
      R_Config_TAU0_1_Start();    // TAU01 動作を有効にします。
      R_Config_TAU0_1_Set_SoftwareTriggerOn();    // A/D 変換の開始トリガを生成します。

      while( 1 ){
            if( interrupt_flag == 1 ) {    // A/D 割り込み発生したかチェックします。
                  interrupt_flag = 0;    // 割り込みフラグをリセットします。
                  continues_num ++;    // 変換カウンタをインクリメントします。
                  // チャンネル 3 から A/D 結果を取得し、バッファに保存します。
                  R_Config_S12AD0_Get_ValueResult (ADCHANNEL3, &AD_buffer_3);
                  //  チャンネル 28 から A/D 結果を取得し、バッファに保存します。
                  R_Config_S12AD0_Get_ValueResult (ADCHANNEL28, &AD_buffer_28);

                  if( continues_num >=3 ) {    // 3 回の変換後に停止します。
                        R_Config_S12AD0_Stop();    // A/D モジュールを停止します。
                        break;
                  }
            }
       };
      interrupt_flag = 2;    // 完了を示すフラグを設定します

      while(1);
}
```

Config_S12AD0_user.c

```
/* Start user code for global. Do not edit comment generated here */
volatile uint8_t interrupt_flag;
/* End user code. Do not edit comment generated here */

static void _near r_Config_S12AD0_interrupt(void)
{
      /* Start user code for r_Config_S12AD0_interrupt. Do not edit comment generated here */
      interrupt_flag = 1;    // 割り込みフラグを設定して、A/D 変換が完了したことを示します。
      /* End user code. Do not edit comment generated here */
}
```

4.2.43 12 ビット A/D グループ・スキャン

以下に、スマート・コンフィグレータが 12 ビット A/D グループ・スキャン用として出力する API 関数の一覧を示します。

表 4.48 12 ビット A/D 連続スキャン用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_S12AD*n*}_Create	12 ビット A/D コンバータ	12 ビット A/D コンバータ・モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_S12AD*n*}_Start		12 ビット A/D コンバータを起動します。
R_{Config_S12AD*n*}_Stop		12 ビット A/D コンバータを停止します。
R_{Config_S12ADn}_Get_ValueResult		バッファに 12 ビットの変換結果を返します。
R_{Config_S12AD*n*}_Create_UserInit		12 ビット A/D コンバータに関するユーザ独自の初期化処理を実行します。
r_{Config_S12AD*n*}_interrupt		INTAD 割り込みに伴う処理を実行します。
r_{Config_S12ADn}_groupb_interrupt		INTADGB 割り込みに伴う処理を実行します。

R_{Config_S12AD*n*}_Create

12 ビット A/D コンバータ・モジュールを制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_S12AD n}_Create(void);
```

備考　　*n* = 0

[引数]

なし

[戻り値]

なし

R_{Config_S12AD*n*}_Start

12 ビット A/D コンバータを起動します。

[指定形式]

```
void    R_{Config_S12ADn}_Start(void);
```

備考　　*n* = 0

[引数]

なし

[戻り値]

なし

R_{Config_S12AD*n*}_Stop

12 ビット A/D コンバータを停止します。

[指定形式]

void R_{Config_S12AD*n*}_Stop(void);

備考 *n* = 0

[引数]

なし

[戻り値]

なし

R_{Config_S12AD*n*}_Get_ValueResult

バッファに 12 ビットの変換結果を返します。

[指定形式]

```
void    R_{Config_S12AD n}_Get_ValueResult (e_ad_channel_t channel, uint16_t * const buffer);
```

備考　　*n* = 0

[引数]

I/O	引数	説明
I	e_ad_channel_t channel;	読み取るデータレジスタのチャネル
I	uint16_t * const buffer;	変換結果を格納するアドレス

備考　　以下に、構造 e_ad_channel_t チャネル（チャネル条件）を示します。

```
typedef enum
{
ADCHANNEL0, ADCHANNEL1, ADCHANNEL2, ADCHANNEL3, ADCHANNEL4,
ADCHANNEL5, ADCHANNEL6, ADCHANNEL7, ADCHANNEL8, ADCHANNEL9,
ADCHANNEL10, ADCHANNEL11, ADCHANNEL12, ADCHANNEL13, ADCHANNEL14,
ADCHANNEL15, ADCHANNEL16, ADCHANNEL17, ADCHANNEL18, ADCHANNEL19,
ADCHANNEL20, ADCHANNEL21, ADCHANNEL22, ADCHANNEL23, ADCHANNEL24,
ADCHANNEL25, ADCHANNEL26, ADCHANNEL27, ADCHANNEL28, ADCHANNEL29,
ADCHANNEL30, ADINTERREFVOLT, ADSELFDIAGNOSIS
} e_ad_channel_t;
```

[戻り値]

なし

R_{Config_S12AD*n*}_Create_UserInit

12 ビット A/D コンバータに関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_S12AD*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_S12ADn}_Create_UserInit(void);
```

備考　*n* = 0

[引数]

なし

[戻り値]

なし

r_{Config_S12AD*n*}_interrupt

INTAD 割り込みに伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_S12AD n}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_S12AD n}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_S12AD n}_interrupt(void);
```

備考　　*n* = 0

[引数]

なし

[戻り値]

なし

r_{Config_S12AD*n*}_groupb_interrupt

INTADGB 割り込みに伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_S12AD n}_groupb_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_S12AD n}_groupb_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_S12AD n}_groupb_interrupt(void);
```

備考　　*n* = 0

[引数]

なし

[戻り値]

なし

使用例

タイマ機能からの同期トリガによりグループ・スキャン・モードの 12 ビット A/D 変換結果を取得する例です。

main.c (1/2)

```
#include "r_cg_macrodriver.h"

void main(void);

extern volatile uint8_t interrupt_flag_GA;    // グループ A 割り込みフラグです。
extern volatile uint8_t interrupt_flag_GB;    // グループ B 割り込みフラグです。
// 3 回の読み取りの A/D 結果を保存する変数を宣言します。
uint16_t GB_ANI0_1;    // グループ B、チャネル ANI0、1 回目の読み取りです。
uint16_t GB_ANI1_1;    // グループ B、チャネル ANI1、1 回目の読み取りです。
uint16_t GB_ANI2_1;    // グループ B、チャネル ANI2、1 回目の読み取りです。
uint16_t GA_ANI3_1;    // グループ A、チャネル ANI3、1 回目の読み取りです。
uint16_t GA_ANI4_1;    // グループ A、チャネル ANI4、1 回目の読み取りです。

uint16_t GB_ANI0_2;    // グループ B、チャネル ANI0、2 回目の読み取りです。
uint16_t GB_ANI1_2;    // グループ B、チャネル ANI1、2 回目の読み取りです。
uint16_t GB_ANI2_2;    // グループ B、チャネル ANI2、2 回目の読み取りです。
uint16_t GA_ANI3_2;    // グループ A、チャネル ANI3、2 回目の読み取りです。
uint16_t GA_ANI4_2;    // グループ A、チャネル ANI4、2 回目の読み取りです。

uint16_t GB_ANI0_3;    // グループ B、チャネル ANI0、3 回目の読み取りです。
uint16_t GB_ANI1_3;    // グループ B、チャネル ANI1、3 回目の読み取りです。
uint16_t GB_ANI2_3;    // グループ B、チャネル ANI2、3 回目の読み取りです。
uint16_t GA_ANI3_3;    // グループ A、チャネル ANI3、3 回目の読み取りです。
uint16_t GA_ANI4_3;    // グループ A、チャネル ANI4、3 回目の読み取りです。

void main(void)
{
    EI();
    RAMSAR = 0x9F;    // RAM へのアクセスを許可します。

    interrupt_flag_GA = 0;
    interrupt_flag_GB = 0;

    // すべての A/D 結果バッファを 0 に初期化します。
    GB_ANI0_1 = 0;    // グループ B、ANI0 バッファ、1 回目の読み取りです。
    GB_ANI1_1 = 0;
    GB_ANI2_1 = 0;
    GA_ANI3_1 = 0;    // グループ A、ANI3 バッファ、1 回目の読み取りです。
    GA_ANI4_1 = 0;
    GB_ANI0_2 = 0;    // グループ B、ANI0 バッファ、2 回目の読み取りです。
    GB_ANI1_2 = 0;
    GB_ANI2_2 = 0;
    GA_ANI3_2 = 0;    // グループ A、ANI3 バッファ、2 回目の読み取りです。
    GA_ANI4_2 = 0;

    R_Config_S12AD0_Start();    // A/D 変換を開始します。
    R_Config_TAU0_0_Start();    // TAU00 ワンショットを有効にします。
    R_Config_TAU0_4_Start();;    // TAU04 ワンショットを有効にします。
    R_Config_TRD0_Start();
    R_Config_TAU0_4_Set_SoftwareTriggerOn();    // トリガ・グループ B です。
```

main.c (1/2)

```
    // 入力電圧 1V
    while(1) {
        // 状態 1：グループ B の変換が完了するまで待機します。
        if(interrupt_flag_GB == 1) {    // 1 回目の読み取り、グループ B の A/D 結果を読み取ります。
            // グループ B（グループ B 変換完了）
            R_Config_S12AD0_Get_ValueResult(ADCHANNEL0, &GB_ANI0_1);
            R_Config_S12AD0_Get_ValueResult(ADCHANNEL1, &GB_ANI1_1);
            R_Config_S12AD0_Get_ValueResult(ADCHANNEL2, &GB_ANI2_1);
            // グループ A（グループ A は開始しない）
            R_Config_S12AD0_Get_ValueResult(ADCHANNEL3, &GA_ANI3_1);
            R_Config_S12AD0_Get_ValueResult(ADCHANNEL4, &GA_ANI4_1);
            interrupt_flag_GB = 2;    // 一度読み取り、グループ B 変換完了。

            R_Config_TAU0_0_Set_SoftwareTriggerOn();    // トリガ・グループ A です。
        }
        // 状態 2：グループ A の変換が完了するまで待機します。
        if(interrupt_flag_GA == 1) {    // 2 回目の読み取りです。
            // グループ B（グループ B は開始しない）
            R_Config_S12AD0_Get_ValueResult(ADCHANNEL0, &GB_ANI0_2);
            R_Config_S12AD0_Get_ValueResult(ADCHANNEL1, &GB_ANI1_2);
            R_Config_S12AD0_Get_ValueResult(ADCHANNEL2, &GB_ANI2_2);
            // グループ A（グループ A は開始します）
            R_Config_S12AD0_Get_ValueResult(ADCHANNEL3, &GA_ANI3_2);
            R_Config_S12AD0_Get_ValueResult(ADCHANNEL4, &GA_ANI4_2);
            interrupt_flag_GA = 2;    // 一度読み取り、グループ A 変換完了。
        }
        // 状態 3：両方のグループが変換を完了したらループを終了します。
        if((interrupt_flag_GA == 2) && (interrupt_flag_GB == 2)) {
            break;    // 全ての変換を完了する。
        }
    }
    R_Config_S12AD0_Stop();  // 変換を停止します。
    // 3 回目の読み取りを準備します。
    GB_ANI0_3 = 0;
    GB_ANI1_3 = 0;
    GB_ANI2_3 = 0;
    GA_ANI3_3 = 0;
    GA_ANI4_3 = 0;
    // グループ A および B フラグをリセットします。
    interrupt_flag_GA = 0;
    interrupt_flag_GB = 0;
    // 入力電圧 0.6V
    R_Config_S12AD0_Start();    // グループ・スキャンをリスタートする。
    R_Config_TAU0_0_Set_SoftwareTriggerOn();    // グループ A を再トリガします。

    while(interrupt_flag_GA != 1);
    // グループ B（グループ B は開始しない）
    R_Config_S12AD0_Get_ValueResult(ADCHANNEL0, &GB_ANI0_3);
    R_Config_S12AD0_Get_ValueResult(ADCHANNEL1, &GB_ANI1_3);
    R_Config_S12AD0_Get_ValueResult(ADCHANNEL2, &GB_ANI2_3);
    // グループ A（グループ A は開始します）
    R_Config_S12AD0_Get_ValueResult(ADCHANNEL3, &GA_ANI3_3);
    R_Config_S12AD0_Get_ValueResult(ADCHANNEL4, &GA_ANI4_3);

    while(1);
}
```

Config_S12AD0_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t  interrupt_flag_GA;
uint8_t  interrupt_flag_GB;
/* End user code. Do not edit comment generated here */

static void __near r_Config_S12AD0_interrupt(void)
{
      /* Start user code for r_Config_S12AD0_interrupt. Do not edit comment generated here */
   interrupt_flag_GA = 1;    // 変換完了時にグループ A 割り込みフラグを設定します。
   /* End user code. Do not edit comment generated here */
}
static void __near r_Config_S12AD0_groupb_interrupt(void)
{
      /* Start user code for r_Config_S12AD0_groupb_interrupt. Do not edit comment generated here */
   interrupt_flag_GB = 1;    // 変換完了時にグループ B 割り込みフラグを設定します。
   /* End user code. Do not edit comment generated here */
}
```

4.2.44 D/A コンバータ

以下に、スマート・コンフィグレータが D/A コンバータ用として出力する API 関数の一覧を示します。

表 4.49 D/A コンバータ用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_DAC*n*}_Create	D/A コンバータ	DAC*n* モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_DAC*n*}_Start		DAC*n* を起動します。
R_{Config_DAC*n*}_Stop		DAC*n* を停止します。
R_{Config_DAC*n*}_Set_ConversionValue		DAC*n* 値を設定します。
R_{Config_DAC*n*}_Create_UserInit		DAC*n* に関するユーザ独自の初期化処理を実行します。

R_{Config_DAC*n*}_Create

DAC*n* モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_DAC_Create から呼び出されます。

[指定形式]

```
void    R_{Config_DACn}_Create(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_DAC*n*}_Start

DAC*n* コンバータを起動します。

[指定形式]

```
void    R_{Config_DACn}_Start(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_DAC*n*}_Stop

DAC*n* コンバータを停止します。

[指定形式]

```
void    R_{Config_DACn}_Stop(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_DAC*n*}_Set_ConversionValue

DAC*n* 値を設定します。

[指定形式]

void R_{Config_DAC*n*}_Set_ConversionValue(uint8_t reg_value);

備考 *n* はチャネル番号を示します。

[引数]

I/O	引数	説明
I	uint8_t reg_value;	D/A 変換を行うデータ

[戻り値]

なし

R_{Config_DAC*n*}_Create_UserInit

DAC*n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_DAC*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_DAC n}_Create_UserInit(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

使用例

D/A 変換を開始する例です。

main.c

```
#include "r_smc_entry.h"

void main(void);

void main(void)
{
    EI();    // 割り込みを有効にします。
    R_Config_DAC0_Set_ConversionValue(0xF0);    // DAC0 の変換値を 0xF0 に設定します。
    R_Config_DAC0_Start();    // D/A 変換を開始します。
}
```

4.2.45 データ・トランスファ・コントローラ

以下に、スマート・コンフィグレータがデータ・トランスファ・コントローラ用として出力する API 関数の一覧を示します。

表 4.50 データ・トランスファ・コントローラ用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_DTC}_Create	データ・トランスファ・コントローラ	DAC モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_DTCD*n*}_Start		DTCDn モジュールの動作を開始します。
R_{Config_DAC*n*}_Stop		DTCDn モジュールの動作を終了します。
R_{Config_DTC}_Create_UserInit		データ・トランスファ・コントローラに関するユーザ独自の初期化処理を実行します。

R_{Config_DTC}_Create

DAC モジュールを制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_DTC}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_DTCD*n*}_Start

DTCD*n* モジュールの動作を開始します。

[指定形式]

```
void    R_{Config_DTCDn}_Start(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_DTCD*n*}_Stop

DTCD*n* モジュールの動作を終了します。

[指定形式]

```
void    R_{Config_DTCDn}_Stop(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_DTC}_Create_UserInit

データ・トランスファ・コントローラに関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_DTC}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_DTC}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

使用例

リアルタイム・クロックのアラーム一致検出の定周期信号に応じて DTC データ転送を開始する例です。

main.c

```
#include "r_smc_entry.h"

void main(void);

void main(void)
{
    EI();   // グローバル割り込みを有効にして、システムが割り込みイベントに応答できるようにします。
    R_DTCD0_Start();    // DTC チャネル 0 を開始して、RTC／アラーム一致検出によってトリガされる自動データ転送を有効にします。
    R_Config_RTC_Start();    // リアルタイム・クロック・モジュールを起動して、定周期信号の生成を開始します。

    while(dtc_controldata_0.dtcct != 0);    // DTC 転送が完了するまで待ちます。（DTCCT は 0 になります）

    R_Config_RTC_Stop();   // 転送完了後にリアルタイム・クロック・モジュールを停止します。
    R_DTCD0_Stop();   // DTC チャネル 0 を停止してデータ転送を終了します。
}
```

4.2.46 コンパレータ

以下に、スマート・コンフィグレータがコンパレータ用として出力する API 関数の一覧を示します。

表 4.51 コンパレータ用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_COMP*n*}_Create	コンパレータ	コンパレータ *n* モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_COMP*n*}_Start		コンパレータ *n* を起動します。
R_{Config_COMP*n*}_Stop		コンパレータ *n* を停止します。
R_{Config_COMP*n*}_Create_UserInit		コンパレータ *n* に関するユーザ独自の初期化処理を実行します。
r_{Config_COMP*n*}_interrupt		INTCMP*n* 割り込みに伴い処理を実行します。

R_{Config_COMP*n*}_Create

コンパレータ *n* モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_COMP_Create から呼び出されます。

[指定形式]

```
void    R_{Config_COMPn}_Create(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_COMP*n*}_Start

コンパレータ *n* を起動します。

[指定形式]

```
void    R_{Config_COMPn}_Start(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_COMP*n*}_Stop

コンパレータ *n* を停止します。

[指定形式]

```
void    R_{Config_COMPn}_Stop(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_COMP*n*}_Create_UserInit

コンパレータ *n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_COMP*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_COMPn}_Create_UserInit(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_COMP*n*}_interrupt

INTCMP*n* 割り込みに伴い処理を実行します。

備考　この API 関数は、コンパレータ割り込みの割り込みハンドラとして呼び出され、コンパレータ出力の有効エッジを検出したときに発生します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_COMPn}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_COMPn}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_COMPn}_interrupt(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

使用例

コンパレータ出力の有効エッジを検出したときにフラグを設定する例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t comp0_trig_flag;
void main(void);

void main(void)
{
    EI();    // グローバル割り込みを有効にして、システムが割り込みイベントに応答できるようにします。
    comp0_trig_flag = 0U;
    /* Start comparator 0 */
    R_Config_COMP0_Start ();    // COMP0 動作を有効にします。
    while(comp0_trig_flag != 1U);    // 有効エッジを待ちます。
    comp1_trig_flag = 0U;
    R_Config_COMP0_Stop ();    // COMP0 動作を無効にします。
}
```

Config_COMP0_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t comp0_trig_flag = 0U;
/* End user code. Do not edit comment generated here */

static void __near r_Config_COMP0_interrupt(void)
{
    /* Start user code for r_Config_COMP0_interrupt. Do not edit comment generated here */
    /* Set the flag */
    comp0_trig_flag = 1U;    // コンパレータ出力の有効エッジが検出されたことを示します。
    /* End user code. Do not edit comment generated here */
}
```

4.2.47 プログラマブル・ゲイン・アンプ

以下に、スマート・コンフィグレータがプログラマブル・ゲイン・アンプ用として出力する API 関数の一覧を示します。

表 4.52 コンパレータ用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_PGA}_Create	コンパレータ	PGA モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_PAG}_Start		PGA を起動します。
R_{Config_PAG}_Stop		PGA を停止します。
R_{Config_PGA}_Create_UserInit		PGA に関するユーザ独自の初期化処理を実行します。

R_{Config_PGA}_Create

PAG モジュールを制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、R_PGACOMP_Create から呼び出されます。

[指定形式]

```
void    R_{Config_PGA}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_PAG}_Start

PAG を起動します。

[指定形式]

```
void    R_{Config_PAG}_Start(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_PAG}_Stop

PAG を停止します。

[指定形式]

```
void    R_{Config_PAG}_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_PGA}_Create_UserInit

PAG に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_PGA}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_PAG}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

使用例

信号を入力するときに増幅信号を出力する例です。

main.c

```
#include "r_smc_entry.h"

void main(void);

void main(void)
{
    EI();    // グローバル割り込みを有効にして、システムが割り込みイベントに応答できるようにします。
    /* Start comparator 0 */
    R_Config_PGA_Start ();
    while(1U);
}
```

4.2.48 SPI(CSI)通信

以下に、スマート・コンフィグレータが SPI(CSI)通信用として出力する API 関数の一覧を示します。

表 4.53 CSI 通信用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_CSI*p*}_Create	シリアル・アレイ・ユニット	CSI*p* モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_CSI*p*}_Start		CSI*p* モジュールの動作を開始します。
R_{Config_CSI*p*}_Stop		CSI*p* モジュールの動作を終了します。
R_{Config_CSI*p*}_Send		CSI*p* データを送信します。
R_{Config_CSI*p*}_Receive		CSI*p* データを受信します。
R_{Config_CSI*p*}_Send_Receive		CSI*p* データを送受信します。
R_{Config_CSI*p*}_Create_UserInit		CSI*p* に関するユーザ独自の初期化処理を実行します。
r_{Config_CSI*p*}_interrupt		バッファ空き／転送完了割り込みに伴う処理を実行します（INTCSI*p*）。
r_{Config_CSI*p*}_callback_sendend		送信完了の検出に伴う処理を実行します。
r_{Config_CSI*p*}_callback_receiveend		受信完了の検出に伴う処理を実行します。
r_{Config_CSI*p*}_callback_error		受信エラーの検出に伴う処理を実行します。

R_{Config_CSI*p*}_Create

CSI*p* モジュールを制御する前に必要な初期化処理を実行します。

備考 1. この API 関数は、R_SAU*m*_Create から呼び出されます。

備考 2. *m*=0 のとき *p*=00, 01, 10, 11
m=1 のとき *p*=20, 21, 30, 31

[指定形式]

```
void    R_{Config_CSIp}_Create(void);
```

備考 *p*=00, 01, 10, 11, 20, 21, 30, 31

[引数]

なし

[戻り値]

なし

R_{Config_CSI*p*}_Start

CSI*p* モジュールの動作を開始します。

[指定形式]

```
void    R_{Config_CSIp}_Start(void);
```

備考　　*p*＝00, 01, 10, 11, 20, 21, 30, 31

[引数]

なし

[戻り値]

なし

R_{Config_CSI*p*}_Stop

CSI*p* モジュールの動作を終了します。

[指定形式]

```
void    R_{Config_CSIp}_Stop(void);
```

備考　　*p*＝00, 01, 10, 11, 20, 21, 30, 31

[引数]

なし

[戻り値]

なし

R_{Config_CSI*p*}_Send

CSI*p* データを送信します。

[指定形式]

```
MD_STATUS    R_{Config_CSIp}_Send(uint8_t * const tx_buf, uint16_t tx_num);
```

備考　　*p*＝00, 01, 10, 11, 20, 21, 30, 31

[引数]

I/O	引数	説明
I	uint8_t * const tx_buf;	送信バッファへのポインタ
I	uint16_t tx_num;	バッファのサイズ

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_ARGERROR	引数の指定が不正

R_{Config_CSI*p*}_Receive

CSI*p* データを受信します。

[指定形式]

```
MD_STATUS    R_{Config_CSIp}_Receive(uint8_t * const rx_buf, uint16_t rx_num);
```

備考　　*p*＝00, 01, 10, 11, 20, 21, 30, 31

[引数]

I/O	引数	説明
O	uint8_t * const rx_buf;	受信バッファへのポインタ
I	uint16_t rx_num;	バッファのサイズ

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_ARGERROR	引数の指定が不正

R_{Config_CSI*p*}_Send_Receive

CSI*p* データを送受信します。

[指定形式]

MD_STATUS R_{Config_CSI*p*}_Send_Receive(uint8_t * const tx_buf, uint16_t tx_num, uint8_t * const rx_buf);

備考 *p*＝00, 01, 10, 11, 20, 21, 30, 31

[引数]

I/O	引数	説明
I	uint8_t * const tx_buf;	送信バッファへのポインタ
O	uint8_t * const rx_buf;	受信バッファへのポインタ
I	uint16_t tx_num;	バッファのサイズ

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_ARGERROR	引数の指定が不正

R_{Config_CSI*p*}_Create_UserInit

CSI*p* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_CSI*p*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_CSIp}_Create_UserInit(void);
```

備考　*p*＝00, 01, 10, 11, 20, 21, 30, 31

[引数]

なし

[戻り値]

なし

r_{Config_CSI*p*}_interrupt

バッファ空き／転送完了割り込みに伴う処理を実行します（INTCSI*p*）。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_CSIp}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_CSIp}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_CSIp}_interrupt(void);
```

備考　　*p*＝00, 01, 10, 11, 20, 21, 30, 31

[引数]

なし

[戻り値]

なし

r_{Config_CSI*p*}_callback_sendend

送信完了の検出に伴う処理を実行します。

備考 1　この API 関数は、CSI*p* 割り込みに伴う処理 r_{Config_CSI*p*}_interrupt のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアのみを保持し、他の処理コードは、コールバックおよび割り込み関数から移動する必要があります。 それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_CSI*p*}_callback_sendend(void);

備考　*p*＝00, 01, 10, 11, 20, 21, 30, 31

[引数]

なし

[戻り値]

なし

r_{Config_CSI*p*}_callback_receiveend

受信完了の検出に伴う処理を実行します。

備考 1　この API 関数は、CSI*p* 割り込みに伴う処理 r_{Config_CSI*p*}_interrupt のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアのみを保持し、他の処理コードは、コールバックおよび割り込み関数から移動する必要があります。 それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_CSI*p*}_callback_receiveend(void);

備考　*p*＝00, 01, 10, 11, 20, 21, 30, 31

[引数]

なし

[戻り値]

なし

r_{Config_CSI*p*}_callback_error

エラー発生の検出に伴う処理を実行します。

備考 1　この API 関数は、CSI*p* 割り込みに伴う処理 r_{Config_CSI*p*}_interrupt のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアのみを保持し、他の処理コードは、コールバックおよび割り込み関数から移動する必要があります。 それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_CSI*p*}_callback_error(uint8_t err_type);

備考　*p*＝00, 01, 10, 11, 20, 21, 30, 31

[引数]

I/O	引数	説明
I	uint8_t err_type;	エラー種別を示す値 Bit0：オーバラン・エラー

[戻り値]

なし

使用例

CSI00 からデータを送信し、CSI11（連続モード）でデータを受信する例です。

main.c

```
#include "r_smc_entry.h"

uint8_t tx_buf[6] = {0xA5,0x3F,0xC0,0x5C,0xB6,0x37};     // 送信する 6 バイトのデータで送信バッファを定義します。
uint8_t rx_buf[6] = {0x00,0x00,0x00,0x00,0x00,0x00};     // 6 バイトの受信データを格納する受信バッファを定義します。
volatile uint8_t transmitend_flag = 0U;
volatile uint8_t receiveend_flag = 0U;

void main(void);

void main(void)
{
    EI();    // グローバル割り込みを有効にして、システムが割り込みイベントに応答できるようにします。
    R_Config_CSI11_Start();    // CS11 モジュール（受信）を起動します。
    R_Config_CSI00_Start();    // CS00 モジュール（送信）を起動します。
    R_Config_CSI11_Receive(rx_buf, sizeof(rx_buf));    // CSI11 でデータの受信を開始し、rx_buf に格納します。
    R_Config_CSI00_Send(tx_buf, sizeof(tx_buf));    // CSI00 でデータの送信を開始し、rx_buf の内容を送信します。
    while(transmitend_flag != 1U);    // 送信が完了するまで待ちます。
    transmitend_flag = 0U;    // 送信完了フラグをリセットします。
    while(receiveend_flag != 1U);    // 受信が完了するまで待ちます。
    receiveend_flag = 0U;    //受信完了フラグをリセットします。
    R_Config_CSI00_Stop();    // 送信完了後に CSI00 モジュールを停止します。
    R_Config_CSI11_Stop();    // 受信完了後に CSI11 モジュールを停止します。
}
```

Config_CSI00_user.c

```
/* Start user code for global. Do not edit comment generated here */
extern volatile uint8_t transmitend_flag;
/* End user code. Do not edit comment generated here */

static void r_Config_CSI00_callback_sendend(void)
{
    /* Start user code for r_Config_CSI00_callback_sendend. Do not edit comment generated here */
    transmitend_flag = 1U;    // 送信完了を示すフラグを設定します。
    /* End user code. Do not edit comment generated here */
}
```

Config_CSI11_user.c

```
/* Start user code for global. Do not edit comment generated here */
extern volatile uint8_t receiveend_flag;
/* End user code. Do not edit comment generated here */

static void r_Config_CSI11_callback_receiveend (void)
{
    /* Start user code for r_Config_CSI11_callback_receiveend. Do not edit comment generated here
*/
    receiveend_flag = 1U;    // 受信完了を示すフラグを設定します。
    /* End user code. Do not edit comment generated here */
}
```

4.2.49 UART 通信（シリアル・アレイ・ユニット）

以下に、スマート・コンフィグレータが UART 通信用として出力する API 関数の一覧を示します。

表 4.54 UART 通信用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_UART*q*}_Create	シリアル・アレイ・ユニット	UART*q* モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_UART*q*}_Start		UART*q* モジュールの動作を開始します。
R_{Config_UART*q*}_Stop		UART*q* モジュールの動作を終了します。
R_{Config_UART*q*}_Send		UART*q* データを送信します。
R_{Config_UART*q*}_Receive		UART*q* データを受信します。
R_{Config_UART*q*}_Loopback_Enable		UART*q* ループバック機能を許可します。
R_{Config_UART*q*}_Loopback_Disable		UART*q* ループバック機能を禁止します。
R_{Config_UART*q*}_Create_UserInit		UART*q* に関するユーザ独自の初期化処理を実行します。
r_{Config_UART*q*}_interrupt_send		UART*q* 送信完了割り込み（シングル転送モード）またはバッファ空き割り込み（連続転送モード）に伴う処理を実行します。
r_{Config_UART*q*}_interrupt_receive		UART*q* 受信完了割り込みに伴う処理を実行します。
r_{Config_UART*q*}_interrupt_error		UART*q* 受信エラーの発生に伴う処理を実行します。
r_{Config_UART*q*}_callback_sendend		送信完了の検出に伴う処理を実行します。
r_{Config_UART*q*}_callback_receiveend		受信完了の検出に伴う処理を実行します。
r_{Config_UART*q*}_callback_error		受信エラーの検出に伴う処理を実行します。
r_{Config_UART*q*}_callback_softwareoverrun		オーバラン・エラーの検出に伴う処理を実行します。

R_{Config_UART*q*}_Create

UART*q* モジュールを制御する前に必要な初期化処理を実行します。

備考 1. この API 関数は、R_SAU*m*_Create から呼び出されます。

備考 2. *m*＝0 のとき *q*＝0, 1
m＝1 のとき *q*＝2, 3

[指定形式]

```
void    R_{Config_UARTq}_Create(void);
```

備考 *q*＝0, 1, 2, 3

[引数]

なし

[戻り値]

なし

R_{Config_UART*q*}_Start

UART*q* モジュールの動作を開始します。

[指定形式]

```
void    R_{Config_UARTq}_Start(void);
```

備考　　*q*＝0, 1, 2, 3

[引数]

なし

[戻り値]

なし

R_{Config_UART*q*}_Stop

UART*q* モジュールの動作を終了します。

[指定形式]

```
void    R_{Config_UARTq}_Stop(void);
```

備考　　*q*＝0, 1, 2, 3

[引数]

なし

[戻り値]

なし

R_{Config_UART*q*}_Send

UART*q* データを送信します。

[指定形式]

MD_STATUS　R_{Config_UART*q*}_Send(uint8_t * const tx_buf, uint16_t tx_num);

備考　　*q*＝0, 1, 2, 3

[引数]

I/O	引数	説明
I	uint8_t * const tx_buf;	送信バッファへのポインタ
I	uint16_t tx_num;	バッファのサイズ

[戻り値]

マクロ	説明
MD_OK	正常終了(最初のデータを送信)
MD_ARGERROR	引数の指定が不正

R_{Config_UART*q*}_Receive

UARTq データを受信します。

[指定形式]

MD_STATUS　R_{Config_UART*q*}_Receive(uint8_t * const rx_buf, uint16_t rx_num);

備考　　*q*＝0, 1, 2, 3

[引数]

I/O	引数	説明
O	uint8_t * const rx_buf;	受信バッファへのポインタ
I	uint16_t rx_num;	バッファのサイズ

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_ARGERROR	引数の指定が不正

R_{Config_UART*q*}_Loopback_Enable

UART*q* ループバック機能を許可します。

[指定形式]

void R_{Config_UART*q*}_Loopback_Enable(void);

備考 *q*＝0, 1, 2, 3

[引数]

なし

[戻り値]

なし

R_{Config_UART*q*}_Loopback_Disable

UART*q* ループバック機能を禁止します。

[指定形式]

void R_{Config_UART*q*}_Loopback_Disable(void);

備考 *q*＝0, 1, 2, 3

[引数]

なし

[戻り値]

なし

R_{Config_UART*q*}_Create_UserInit

UART*q* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_UART*q*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_UART q}_Create_UserInit(void);
```

備考　*q*＝0, 1, 2, 3

[引数]

なし

[戻り値]

なし

r_{Config_UART*q*}_interrupt_send

UART*q* 送信完了割り込み（シングル転送モード）またはバッファ空き割り込み（連続転送モード）に伴う処理を実行します。

［指定形式］

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_UARTq}_interrupt_send(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_UARTq}_interrupt_send(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_UARTq}_interrupt_send(void);
```

備考　　*q*＝0, 1, 2, 3

［引数］

なし

［戻り値］

なし

r_{Config_UARTq}_interrupt_receive

UARTq 受信完了割り込みに伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_UARTq}_interrupt_receive(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_UARTq}_interrupt_receive(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_UARTq}_interrupt_receive(void);
```

備考　　*q*＝0, 1, 2, 3

[引数]

なし

[戻り値]

なし

r_{Config_UARTq}_interrupt_error

UARTq 受信エラーの発生に伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_UARTq}_interrupt_error(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_UARTq}_interrupt_error(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_UARTq}_interrupt_error(void);
```

備考　　q＝0, 1, 2, 3

[引数]

なし

[戻り値]

なし

r_{Config_UARTq}_callback_sendend

送信完了の検出に伴う処理を実行します。

備考 1　この API 関数は、UART*q* 送信完了割り込みに伴う処理 r_{Config_UARTq}_interrupt_send のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアのみを保持し、他の処理コードは、コールバックおよび割り込み関数から移動する必要があります。 それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_UARTq}_callback_sendend(void);

備考　*q*＝0, 1, 2, 3

[引数]

なし

[戻り値]

なし

r_{Config_UARTq}_callback_receiveend

受信完了の検出に伴う処理を実行します。

備考 1　この API 関数は、UART*q* 受信完了割り込みに伴う処理 r_{Config_UARTq}_interrupt_receiveend のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアのみを保持し、他の処理コードは、コールバックおよび割り込み関数から移動する必要があります。 それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_UARTq}_callback_receiveend(void);

備考　*q*＝0, 1, 2, 3

[引数]

なし

[戻り値]

なし

r_{Config_UARTq}_callback_error

受信エラーの検出に伴う処理を実行します。

備考 1　この API 関数は、UARTq 受信エラー割り込みに伴う処理 r_{Config_UARTq}_interrupt_error のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアのみを保持し、他の処理コードは、コールバックおよび割り込み関数から移動する必要があります。 それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_UARTq}_callback_error(uint8_t err_type);

備考　*q*＝0, 1, 2, 3

[引数]

I/O	引数	説明
I	uint8_t err_type;	エラー種別を示す値 Bit 0：オーバラン・エラー Bit 1：フレーミング・エラー Bit 2：パリティ・エラー Bit 3 - 7：0

[戻り値]

なし

r_{Config_UARTq}_callback_softwareoverrun

オーバラン・エラーの検出に伴う処理を実行します。

備考 1　この API 関数は、UARTq 受信完了割り込みに伴う処理 r_{Config_UARTq}_interrupt_receive のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアのみを保持し、他の処理コードは、コールバックおよび割り込み関数から移動する必要があります。 それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_UARTq}_callback_softwareoverrun(uint16_t rx_data);

備考　*q*＝0, 1, 2, 3

[引数]

I/O	引数	説明
I	uint16_t rx_data;	受信データ

[戻り値]

なし

使用例

UART0 からデータを送信し、UART1 でデータを受信する例です。

main.c

```
#include "r_smc_entry.h"

uint8_t tx_buf[6] = {0xA5,0x3F,0xC0,0x5C,0xB6,0x37};     // 送信する 6 バイトのデータで送信バッファを定義します。
uint8_t rx_buf[6] = {0x00,0x00,0x00,0x00,0x00,0x00};     // 6 バイトの受信データを格納する受信バッファを定義します。
volatile uint8_t transmitend_flag = 0U;
volatile uint8_t receiveend_flag = 0U;

void main(void);

void main(void)
{
    EI();    // グローバル割り込みを有効にして、システムが割り込みイベントに応答できるようにします。
    R_Config_UART0_Start();    // UART0 モジュール（送信）を起動します。
    R_Config_UART1_Start();    // UART1 モジュール（受信）を起動します。
    R_Config_UART1_Receive(rx_buf, 6);      // UART1 でデータの受信を開始し、rx_buf に格納します。
    R_Config_UART0_Send(tx_buf, 6);     // UART0 でデータの送信を開始し、rx_buf の内容を送信します。

    while(transmitend_flag != 1U && receiveend_flag != 1U);     // 送信と受信の両方が完了するまで待ちます。
    transmitend_flag = 0U;    // 送信完了フラグをリセットします。
    receiveend_flag = 0U;    // 受信完了フラグをリセットします。
    R_Config_UART0_Stop();    // 送信完了後に UART0 モジュールを停止します。
    R_Config_UART1_Stop();    // 受信完了後に UART1 モジュールを停止します。
}
```

Config_UART0_user.c

```
/* Start user code for global. Do not edit comment generated here */
extern volatile uint8_t transmitend_flag;
/* End user code. Do not edit comment generated here */

static void r_Config_UART0_callback_sendend(void)
{
    /* Start user code for r_Config_UART0_callback_sendend. Do not edit comment generated here */
    transmitend_flag = 1U;    // 送信完了を示すフラグを設定します。
    /* End user code. Do not edit comment generated here */
}
```

Config_UART1_user.c

```
/* Start user code for global. Do not edit comment generated here */
extern volatile uint8_t receiveend_flag;
/* End user code. Do not edit comment generated here */

static void r_Config_UART1_callback_receiveend (void)
{
    /* Start user code for r_Config_UART1_callback_receiveend. Do not edit comment generated here
*/
    receiveend_flag = 1U;    // 受信完了を示すフラグを設定します。
    /* End user code. Do not edit comment generated here */
}
```

4.2.50 UART 通信（シリアル・インタフェース UARTA）

以下に、スマート・コンフィグレータが UART 通信の UARTA 用として出力する API 関数の一覧を示します。

表 4.55 UART 通信（UARTA）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_UARTA*n*}_Create	シリアル・インタフェース UARTA	UARTA*n* モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_UARTA*n*}_Start		UARTA*n* モジュールの動作を開始します。
R_{Config_UARTA*n*}_Stop		UARTA*n* モジュールの動作を終了します。
R_{Config_UARTA*n*}_Send		UARTA*n* データを送信します。
R_{Config_UARTA*n*}_Receive		UARTA*n* データを受信します。
R_{Config_UARTA*n*}_Loopback_Enable		UARTA*n* ループバック機能を許可します。
R_{Config_UARTA*n*}_Loopback_Disable		UARTA*n* ループバック機能を禁止します。
R_{Config_UARTA*n*}_Create_UserInit		UARTA*n* に関するユーザ独自の初期化処理を実行します。
R_{Config_UARTA*n*}_PollingEnd_UserCode		ポーリングによる連続送信の完了に応じて、ユーザコードを実行します。
r_{Config_UARTA*n*}_interrupt_send		UARTA*n* 送信信完了割り込み（INTUT*n*）に伴う処理を実行します。
r_{Config_UARTA*n*}_interrupt_receive		UARTA*n* 受信完了割り込み（INTUR*n*）に伴う処理を実行します。
r_{Config_UARTA*n*}_interrupt_error		UARTA*n* 受信エラー割り込み（INTURE*n*）に伴う処理を実行します。
r_{Config_UARTA*n*}_callback_sendend		送信完了の検出に伴う処理を実行します。
r_{Config_UARTA*n*}_callback_receiveend		受信完了の検出に伴う処理を実行します。
r_{Config_UARTA*n*}_callback_error		受信エラーの検出に伴う処理を実行します。

R_{Config_UARTA*n*}_Create

UARTA*n* モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、R_UARTA_Create から呼び出されます。

[指定形式]

```
void    R_{Config_UARTAn}_Create(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_UARTA*n*}_Start

UARTA*n* モジュールの動作を開始します。

[指定形式]

```
void    R_{Config_UARTAn}_Start(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_UARTA*n*}_Stop

UARTA*n* モジュールの動作を終了します。

[指定形式]

```
void    R_{Config_UARTAn}_Stop(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_UARTA*n*}_Send

UARTA*n* データを送信します。

[指定形式]

MD_STATUS　R_{Config_UARTA*n*}_Send(uint8_t * const tx_buf, uint16_t tx_num);

備考　　*n* はチャネル番号を示します。

[引数]

I/O	引数	説明
I	uint8_t * const tx_buf;	送信バッファへのポインタ
I	uint16_t tx_num;	バッファのサイズ

[戻り値]

マクロ	説明
MD_OK	正常終了(最初のデータを送信)
MD_ARGERROR	引数の指定が不正

R_{Config_UARTA*n*}_Receive

UARTA*n* データを受信します。

[指定形式]

```
MD_STATUS    R_{Config_UARTAn}_Receive(uint8_t * const rx_buf, uint16_t rx_num);
```

備考　　*n* はチャネル番号を示します。

[引数]

I/O	引数	説明
O	uint8_t * const rx_buf;	受信バッファへのポインタ
I	uint16_t tx_num;	バッファのサイズ

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_ARGERROR	引数の指定が不正

R_{Config_UARTA*n*}_Loopback_Enable

UARTA*n* ループバック機能を許可します。

[指定形式]

void R_{Config_UARTA*n*}_Loopback_Enable(void);

備考 *n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_UARTA*n*}_Loopback_Disable

UARTA*n* ループバック機能を禁止します。

[指定形式]

void R_{Config_UARTA*n*}_Loopback_Disable(void);

備考 *n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_UARTA*n*}_Create_UserInit

UARTA*n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_UARTA*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_UARTAn}_Create_UserInit(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_UARTA*n*}_PollingEnd_UserCode

ポーリングによる連続送信の完了に応じて、ユーザコードを実行します。

備考　この API 関数は、データ送信の完了に対応する R_{Config_UARTA*n*}_Send から呼び出されます。

[指定形式]

```
void    R_{Config_UARTAn}_PollingEnd_UserCode(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_UARTA*n*}_interrupt_send

UART*n* 送信信完了割り込み（INTUT*n*）に伴う処理を実行します。

［指定形式］

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_UARTAn}_interrupt_send(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_UARTAn}_interrupt_send(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_UARTAn}_interrupt_send(void);
```

備考　　*n* はチャネル番号を示します。

［引数］

なし

［戻り値］

なし

r_{Config_UARTA*n*}_interrupt_receive

UARTA*n* 受信完了割り込み（INTUR*n*）に伴う処理を実行します。

［指定形式］

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_UARTAn}_interrupt_receive(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_UARTAn}_interrupt_receive(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_UARTAn}_interrupt_receive(void);
```

備考　　*n* はチャネル番号を示します。

［引数］

なし

［戻り値］

なし

r_{Config_UARTA*n*}_interrupt_error

UARTA*n* 受信エラー割り込み（INTURE*n*）に伴う処理を実行します。

［指定形式］

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_UARTAn}_interrupt_error(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_UARTAn}_interrupt_error(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_UARTAn}_interrupt_error(void);
```

備考　　*n* はチャネル番号を示します。

［引数］

なし

［戻り値］

なし

r_{Config_UARTA*n*}_callback_sendend

送信完了の検出に伴う処理を実行します。

備考 1　この API 関数は、UARTA*n* 送信完了割り込みに伴う処理 r_{Config_UARTA*n*}_interrupt_send のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアのみを保持し、他の処理コードは、コールバックおよび割り込み関数から移動する必要があります。 それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_UARTA*n*}_callback_sendend(void);

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_UARTA*n*}_callback_receiveend

受信完了の検出に伴う処理を実行します。

備考 1　この API 関数は、UARTA*n* 受信完了割り込みに伴う処理 r_{Config_UARTA*n*}_interrupt_receive のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアのみを保持し、他の処理コードは、コールバックおよび割り込み関数から移動する必要があります。 それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_UARTA*n*}_callback_receiveend(void);

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_UARTA*n*}_callback_error

受信エラーの検出に伴う処理を実行します。

備考 1　この API 関数は、UARTA*n* 受信エラー割り込みに伴う処理 r_{Config_UARTA*n*}_interrupt_error のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアのみを保持し、他の処理コードは、コールバックおよび割り込み関数から移動する必要があります。 それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_UARTA*n*}_callback_error(uint8_t err_type);

備考　*n* はチャネル番号を示します。

[引数]

I/O	引数	説明
I	uint8_t err_type;	エラー種別を示す値 Bit 0：オーバラン・エラー Bit 1：フレーミング・エラー Bit 2：パリティ・エラー Bit 3 - 7：0

[戻り値]

なし

使用例

UARTA0 からポーリング・モードでデータを送信し、UARTA1 もデータを 2 回送受信する例です。

main.c

```
#include "r_smc_entry.h"

uint8_t tx_buf0[] = {0x7A};    // UARTA0 の送信バッファ（1 byte）
uint8_t tx_buf1[] = {0x7A, 0x6C, 0x27, 0x1F, 0xF8};    // UARTA1 の送信バッファ（5 byte）
uint8_t rx_buf0[] = {0x00};    // UARTA0 の受信バッファ（1 byte）
uint8_t rx_buf1[] = {0x00, 0x00, 0x00, 0x00, 0x00};    // UARTA1 の受信バッファ（5 byte）
volatile uint8_t transmitend_flag = 0U;
volatile uint8_t receptend_flag = 0U;
void main(void);

void main(void)
{
    EI();    // グローバル割り込みを有効にして、システムが割り込みイベントに応答できるようにします。
    R_Config_UARTA0_Start();    // UARTA0 モジュールを起動します。
    R_Config_UARTA1_Start();    // UARTA1 モジュールを起動します。

    // 1 回目の送信（UARTA0 から UARTA1)
    R_Config_UARTA1_Receive(rx_buf0, sizeof(rx_buf0));    // UARTA1 は 1 バイトの受信を準備します。
    R_Config_UARTA0_Send(tx_buf0, sizeof(tx_buf0));    // UARTA0 は 1 バイトを送信します。

    while((1U != transmitend_flag) || (1U != receptend_flag));    // 送信と受信の両方が完了するまで待ちます。

    // 2 回目の送信（UARTA1 から UARTA0)
    R_Config_UARTA0_Receive(rx_buf1, sizeof(rx_buf1));    // UARTA0 は 5 バイトの受信を準備します。
    R_Config_UARTA1_Send(tx_buf1, sizeof(tx_buf1));    // UARTA1 は 5 バイトを送信します。

    while((2U != transmitend_flag) || (2U != receptend_flag));    // 送信と受信の両方が完了するまで待ちます。

    R_Config_UARTA0_Stop();    // UARTA0 モジュールを停止します。
    R_Config_UARTA1_Stop();    // UARTA1 モジュールを停止します。
}
```

Config_UARTA0_user.c

```
/* Start user code for global. Do not edit comment generated here */
extern volatile uint8_t transmitend_flag;
extern volatile uint8_t receptend_flag;
/* End user code. Do not edit comment generated here */

void R_Config_UARTA0_PollingEnd_UserCode(void)
{
    /* Start user code for R_Config_UARTA0_PollingEnd_UserCode. Do not edit comment generated here */
    ransmitend_flag++;    // UARTA0 送信が完了したら送信フラグをインクリメントします。
    /* End user code. Do not edit comment generated here */
}

static void r_Config_UARTA0_callback_receiveend (void)
{
    /* Start user code for r_Config_UARTA0_callback_sendend. Do not edit comment generated here */
    receptend_flag++;    // UARTA0 受信が完了したら受信フラグをインクリメントします。
    /* End user code. Do not edit comment generated here */
}
```

Config_UARTA1_user.c

```
/* Start user code for global. Do not edit comment generated here */
extern volatile uint8_t transmitend_flag;
extern volatile uint8_t receptend_flag;
/* End user code. Do not edit comment generated here */

void R_Config_UARTA1_PollingEnd_UserCode (void)
{
    /* Start user code for R_Config_UARTA1_PollingEnd_UserCode. Do not edit comment generated here */
    transmitend_flag++;    // UARTA1 送信が完了したら送信フラグをインクリメントします。
    /* End user code. Do not edit comment generated here */
}

static void r_Config_UARTA1_callback_receiveend (void)
{
    /* Start user code for r_Config_UARTA1_callback_receiveend. Do not edit comment generated here */
    receptend_flag++;    // UARTA1 受信が完了したら受信フラグをインクリメントします。
    /* End user code. Do not edit comment generated here */
}
```

4.2.51 UART 通信（LIN/UART モジュール）

以下に、スマート・コンフィグレータが UART 通信の LIN/UART モジュール用として出力する API 関数の一覧を示します。

表 4.56 UART 通信（LIN/UART モジュール）用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_RLIN3*n*}_Create	LIN/UART	RLIN3*n* モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_RLIN3*n*}_Start		RLIN3*n* モジュールの動作を開始します。
R_{Config_RLIN3*n*}_Stop		RLIN3*n* モジュールの動作を終了します。
R_{Config_RLIN3*n*}_Send		RLIN3*n* データを送信します。
R_{Config_RLIN3*n*}_Receive		RLIN3*n* データを受信します。
R_{Config_RLIN3*n*}_Create_UserInit		RLIN3*n* に関するユーザ独自の初期化処理を実行します。
r_{Config_RLIN3*n*}_interrupt_send		LIN3*n* 送信終了割り込み（シングル転送モードの場合）またはバッファエンプティ割り込み（連続転送モードの場合）に伴う処理を実行します。
r_{Config_RLIN3*n*}_interrupt_receive		RLIN3*n* 受信完了割り込みに伴う処理を実行します。
r_{Config_RLIN3*n*}_interrupt_error		RLIN3*n* 受信エラー割り込みに伴う処理を実行します。
r_{Config_RLIN3*n*}_callback_sendend		送信完了の検出に伴う処理を実行します。
r_{Config_RLIN3*n*}_callback_receiveend		受信完了の検出に伴う処理を実行します。
r_{Config_RLIN3*n*}_callback_error		受信エラーの検出に伴う処理を実行します。

R_{Config_RLIN3*n*}_Create

RLIN3*n* モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_RLIN3n}_Create(void);
```

備考　*n* はチャネル番号 (*n* = 0, 1, 2) を示します。

[引数]

なし

[戻り値]

なし

R_{Config_RLIN3*n*}_Start

RLIN3*n* モジュールの動作を開始します。

[指定形式]

void R_{Config_RLIN3*n*}_Start(void);

備考 *n* はチャネル番号 (*n* = 0, 1, 2) を示します。

[引数]

なし

[戻り値]

なし

R_{Config_RLIN3*n*}_Stop

RLIN3*n* モジュールの動作を終了します。

[指定形式]

void R_{Config_RLIN3*n*}_Stop(void);

備考 *n* はチャネル番号 (*n* = 0, 1, 2) を示します。

[引数]

なし

[戻り値]

なし

R_{Config_RLIN3*n*}_Send

RLIN3*n* データを送信します。

[指定形式]

MD_STATUS　R_{Config_RLIN3*n*}_Send(uint8_t * const tx_buf, uint16_t tx_num);

備考　　*n* はチャネル番号 (*n* = 0, 1, 2) を示します。

[引数]

I/O	引数	説明
I	uint8_t * const tx_buf;	送信バッファへのポインタ
I	uint16_t tx_num;	バッファのサイズ

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_ARGERROR	引数の指定が不正

R_{Config_RLIN3*n*}_Receive

RLIN3*n* データを受信します。

[指定形式]

MD_STATUS　R_{Config_RLIN3*n*}_Receive(uint8_t * const rx_buf, uint16_t rx_num);

備考　　*n* はチャネル番号 (*n* = 0, 1, 2) を示します。

[引数]

I/O	引数	説明
O	uint8_t * const rx_buf;	受信バッファへのポインタ
I	uint16_t tx_num;	バッファのサイズ

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_ARGERROR	引数の指定が不正

R_{Config_RLIN3*n*}_Create_UserInit

RLIN3*n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_RLIN3*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_RLIN3n}_Create_UserInit(void);
```

備考　*n* はチャネル番号 (*n* = 0, 1, 2) を示します。

[引数]

なし

[戻り値]

なし

r_{Config_RLIN3*n*}_interrupt_send

RLIN3*n* 送信終了割り込み（シングル転送モードの場合）またはバッファエンプティ割り込み（連続転送モードの場合）に伴う処理を実行します。

［指定形式］

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_RLIN3n}_interrupt_send(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_RLIN3n}_interrupt_send(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_RLIN3n}_interrupt_send(void);
```

備考　　*n* はチャネル番号 (*n* = 0, 1, 2) を示します。

［引数］

なし

［戻り値］

なし

r_{Config_RLIN3*n*}_interrupt_receive

RLIN3*n* 受信完了割り込みに伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_RLIN3n}_interrupt_receive(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_RLIN3n}_interrupt_receive(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_RLIN3n}_interrupt_receive(void);
```

備考　　*n* はチャネル番号 (*n* = 0, 1, 2) を示します。

[引数]

なし

[戻り値]

なし

r_{Config_RLIN3*n*}_interrupt_error

RLIN3*n* 受信エラー割り込みに伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_RLIN3n}_interrupt_error(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_RLIN3n}_interrupt_error(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_RLIN3n}_interrupt_error(void);
```

備考　　*n* はチャネル番号 (*n* = 0, 1, 2) を示します。

[引数]

なし

[戻り値]

なし

r_{Config_RLIN3*n*}_callback_sendend

送信完了の検出に伴う処理を実行します。

備考 1　この API 関数は、RLIN3*n* 送信完了割り込みに伴う処理 r_{Config_RLIN3*n*}_interrupt_send のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアのみを保持し、他の処理コードは、コールバックおよび割り込み関数から移動する必要があります。 それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_RLIN3*n*}_callback_sendend(void);

備考　*n* はチャネル番号 (*n* = 0, 1, 2) を示します。

[引数]

なし

[戻り値]

なし

r_{Config_RLIN3*n*}_callback_receiveend

受信完了の検出に伴う処理を実行します。

備考 1　この API 関数は、RLIN3*n* 受信完了割り込みに伴う処理 r_{Config_RLIN3*n*}_interrupt_receive のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアのみを保持し、他の処理コードは、コールバックおよび割り込み関数から移動する必要があります。 それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_RLIN3*n*}_callback_receiveend(void);

備考　*n* はチャネル番号 (*n* = 0, 1, 2) を示します。

[引数]

なし

[戻り値]

なし

r_{Config_RLIN3*n*}_callback_error

受信エラーの検出に伴う処理を実行します。

備考 1　この API 関数は、RLIN3*n* 受信エラー割り込みに伴う処理 r_{Config_RLIN3*n*}_interrupt_error のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアのみを保持し、他の処理コードは、コールバックおよび割り込み関数から移動する必要があります。 それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_RLIN3*n*}_callback_error(uint8_t err_type);

備考　*n* はチャネル番号 (*n* = 0, 1, 2) を示します。

[引数]

I/O	引数	説明
I	uint8_t err_type;	エラー種別を示す値 Bit 2：オーバラン・エラー Bit 3：フレーミング・エラー Bit 4：拡張ビット検出フラグ Bit 5：ID 一致フラグ Bit 6：パリティ・エラー Bit 0, 1, 7：0

[戻り値]

なし

使用例

RLIN30 がデータを受信し、RLIN31 がデータを送信する例です。

main.c

```
#include "r_smc_entry.h"

uint8_t tx_buf0[] = {0x7a, 0x85, 0xbc, 0x26, 0x01, 0x4f};    // RLIN31 の送信バッファ（送信する 6 バイトのデータ）
uint8_t rx_buf0[] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00};    // RLIN30 の受信バッファ（受信データを格納するための 6 バイトのスペース）
volatile uint8_t sendend_flag = 0U;
volatile uint8_t receiveend_flag = 0U;

void main(void);

void main(void)
{
    EI();    // グローバル割り込みを有効にして、システムが割り込みイベントに応答できるようにします。

    R_Config_RLIN30_Start();    // RLIN30 モジュール（受信）を起動します。
    R_Config_RLIN31_Start();    // RLIN31 モジュール（送信）を起動します。

    sendend_flag = 0U;    // 送信完了フラグをリセットします。
    receiveend_flag = 0U;    // 受信完了フラグをリセットします。
    R_Config_RLIN30_Receive(rx_buf0, sizeof(rx_buf0));    // RLIN30 でデータの受信を開始します。
    R_Config_RLIN31_Send(tx_buf0, sizeof(tx_buf0));    // RLIN31 でデータの送信を開始します。
    while((1U != sendend_flag) || (1U != receiveend_flag));    //

    R_Config_RLIN30_Stop();    // RLIN30 モジュールを停止します。
    R_Config_RLIN31_Stop();    // RLIN31 モジュールを停止します。
    while(1);    // プログラムを実行し続けるためのループ。
}
```

Config_RLIN31_user.c

```
/* Start user code for global. Do not edit comment generated here */
extern volatile uint8_t sendend_flag;
/* End user code. Do not edit comment generated here */

static void r_Config_RLIN31_callback_sendend(void)
{
    /* Start user code for r_Config_RLIN31_callback_sendend. Do not edit comment generated here */
    sendend_flag = 1U;    // RLIN31 の送信完了を示すフラグを設定します。
    /* End user code. Do not edit comment generated here */
}
```

Config_RLIN30_user.c

```
/* Start user code for global. Do not edit comment generated here */
extern volatile uint8_t receiveend_flag;
/* End user code. Do not edit comment generated here */

static void r_Config_RLIN30_callback_receiveend (void)
{
    /* Start user code for r_Config_RLIN30_callback_receiveend. Do not edit comment generated here
*/
    receiveend_flag = 1U;    // RLIN30 の受信完了を示すフラグを設定します。
    /* End user code. Do not edit comment generated here */
}
```

4.2.52 DALI 通信（コントロールデバイス）

以下に、スマート・コンフィグレータが DALI 通信（コントロールデバイス）モジュール用として出力する API 関数の一覧を示します。

表 4.57 DALI 通信（コントロールデバイス）API 関数（1/2）

API 関数名	周辺機能	機能概要
R_{Config_DALI}_Create	DALI	DALI 通信（コントロールデバイス）モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_DALI}_Start		DALI 通信（コントロールデバイス）モジュールの動作を開始します。
R_{Config_DALI}_Stop		DALI 通信（コントロールデバイス）モジュールの動作を終了します。
R_{Config_DALI}_SoftwareReset		DALI 通信（コントロールデバイス）モジュールをリセットします。
R_{Config_DALI}_EnableForceActiveState		DALITxD0 アサートを有効にし、アサートレベル (アクティブ状態) が Low であることを示します。 DALITxD0 端子からの出力を Low にします。
R_{Config_DALI}_DisableForceActiveState		DALITxD0 アサートを無効にします。内部送信データを DALITxD0 端子から出力します。
R_{Config_DALI}_GetStatus		DALI 通信（コントロールデバイス）の状態を取得します。
R_{Config_DALI}_Send		フレームデータを送信します。フレーム長は GUI で設定され、固定値になります。
R_{Config_DALI}_GetReceivedFrame		フレームデータとフレーム長を受信します。
R_{Config_DALI}_Create_UserInit		DALI 通信（コントロールデバイス）に関するユーザ独自の初期化処理を実行します。
r_{Config_DALI}_interrupt_send		DALI 通信（コントロールデバイス）送信時の 32 ビットデータ終了割り込み（INTTD）に伴う処理を実行します。
r_{Config_DALI}_interrupt_receive		DALI 通信（コントロールデバイス）受信時の 32 ビットデータ終了割り込み（INTTD）に伴う処理を実行します。
r_{Config_DALI}_interrupt_error		DALI 通信（コントロールデバイス）の受信エラー割り込み（INTED）に伴う処理を実行します。
r_{Config_DALI}_interrupt_falling_edge_detection		DALI 通信（コントロールデバイス）の立ち下がりエッジ割り込み（INTFED）に伴う処理を実行します。
r_{Config_DALI}_interrupt_power_down_detection		DALI 通信（コントロールデバイス）のパワーダウン割り込み（INTBPD）に伴う処理を実行します。
r_{Config_DALI}_interrupt_collision_detection		DALI 通信（コントロールデバイス）のコリジョン検出割り込み（INTCLD）に伴う処理を実行します。
r_{Config_DALI}_interrupt_stop_bit_detection		DALI 通信（コントロールデバイス）のストップビット検出割り込み（INTSDD）に伴う処理を実行します。

表 4.58 DALI 通信（コントロールデバイス）API 関数（2/2）

API 関数名	周辺機能	機能概要
r_{Config_DALI}_callback_sendend	DALI	送信完了の検出に伴う処理を実行します。
r_{Config_DALI}_callback_receiveend		受信完了の検出に伴う処理を実行します。
r_{Config_DALI}_callback_error		受信エラーの検出に伴う処理を実行します。

R_{Config_DALI}_Create

DALI 通信（コントロールデバイス）モジュールを制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_DALI}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_DALI}_Start

DALI 通信（コントロールデバイス）モジュールの動作を開始します。

[指定形式]

```
void    R_{Config_DALI}_Start(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_DALI}_Stop

DALI 通信（コントロールデバイス）モジュールの動作を終了します。

[指定形式]

```
void    R_{Config_DALI}_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_DALI}_SoftwareReset

DALI 通信（コントロールデバイス）モジュールをリセットします。

[指定形式]

```
void    R_{Config_DALI}_SoftwareReset(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_DALI}_EnableForceActiveState

DALITxD0 アサートを有効にし、アサートレベル (アクティブ状態) が Low であることを示します。DALITxD0 端子からの出力を Low にします。

[指定形式]

```
void    R_{Config_DALI}_ EnableForceActiveState (void);
```

[引数]

なし

[戻り値]

なし

R_{Config_DALI}_DisableForceActiveState

DALITxD0 アサートを無効にします。内部送信データを DALITxD0 端子から出力します。

[指定形式]

```
void    R_{Config_DALI}_ DisableForceActiveState (void);
```

[引数]

なし

[戻り値]

なし

R_{Config_DALI}_GetStatus

DALI 通信（コントロールデバイス）の状態を取得します。

［指定形式］

```
void    R_{Config_DALI}_GetStatus(uint16_t * const status);
```

［引数］

I/O	引数	説明
O	uint16_t * const status;	DALI 状態レジスタバッファへのポインタ

［戻り値］

なし

R_{Config_DALI}_Send

フレームデータを送信します。フレーム長は GUI で設定され、固定値になります。

[指定形式]

```
void    R_{Config_DALI}_Send(uint16_t * const tx_buf);
```

[引数]

I/O	引数	説明
I	uint8_t * const tx_buf;	送信バッファへのポインタ

備考　　以下に、フレーム長と tx_buf 長の関係を示します。

フレーム長	tx_buf[] 長
8 bits	1
16 bits	1
17 bits	2
20 bits	2
24 bits	2
32 bits	2
64 bits	4
128 bits	8
256 bits	16

[戻り値]

なし

R_{Config_DALI}_GetReceivedFrame

フレームデータとフレーム長を受信します。

[指定形式]

```
MD_STATUS R_{Config_DALI}_GetReceivedFrame(uint32_t * const rx_buf, uint16_t * const rx_num);
```

[引数]

I/O	引数	説明
O	uint32_t * const rx_buf;	受信バッファへのポインタ
I	uint16_t tx_num;	バッファフレーム長

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_ARGERROR	引数の指定が不正

R_{Config_DALI}_Create_UserInit

DALI 通信（コントロールデバイス）に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_DALI}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_DALI}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_DALI}_interrupt_send

DALI 通信（コントロールデバイス）送信時の 32 ビットデータ終了割り込み（INTTD）に伴う処理を実行します。

備考　本 API 関数は、送信データ長が 32bit より大きい場合にのみ有効です。

［指定形式］

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_DALI}_interrupt_send(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_DALI}_interrupt_send(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_DALI}_interrupt_send(void);
```

［引数］

なし

［戻り値］

なし

r_{Config_DALI}_interrupt_receive

DALI 通信（コントロールデバイス）受信時の 32 ビットデータ終了割り込み（INTTD）に伴う処理を実行します。

備考　本 API 関数は、送信データ長が 32bit より大きい場合にのみ有効です。

［指定形式］

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_DALI}_interrupt_receive(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_DALI}_interrupt_receive(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_DALI}_interrupt_receive(void);
```

［引数］

なし

［戻り値］

なし

r_{Config_DALI}_interrupt_error

DALI 通信（コントロールデバイス）の受信エラー割り込み（INTED）に伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_DALI}_interrupt_error(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_DALI}_interrupt_error(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_DALI}_interrupt_error(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_DALI}_interrupt_falling_edge_detection

DALI 通信（コントロールデバイス）の立ち下がりエッジ割り込み（INTFED）に伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_DALI}_interrupt_falling_edge_detection(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_DALI}_interrupt_falling_edge_detection(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_DALI}_interrupt_falling_edge_detection(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_DALI}_interrupt_power_down_detection

DALI 通信（コントロールデバイス）のパワーダウン割り込み（INTBPD）に伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_DALI}_interrupt_power_down_detection(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_DALI}_interrupt_power_down_detection(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_DALI}_interrupt_power_down_detection(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_DALI}_interrupt_collision_detection

DALI 通信（コントロールデバイス）のコリジョン検出割り込み（INTCLD）に伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_DALI}_interrupt_collision_detection(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_DALI}_interrupt_collision_detection(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_DALI}_interrupt_collision_detection(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_DALI}_interrupt_stop_bit_detection

DALI 通信（コントロールデバイス）のストップビット検出割り込み（INTSDD）に伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_DALI}_interrupt_stop_bit_detection(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_DALI}_interrupt_stop_bit_detection(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_DALI}_interrupt_stop_bit_detection(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_DALI}_callback_sendend

ストップビット割込みの検出に伴う処理を実行します。

備考 1　この API 関数は、DALI 通信（コントロールデバイス）ストップビット検出割り込みに伴う処理 r_{Config_DALI}_interrupt_stop_bit_detection のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアのみを保持し、他の処理コードは、コールバックおよび割り込み関数から移動する必要があります。 それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

```
static void    r_{Config_DALI}_callback_sendend(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_DALI}_callback_receiveend

ストップビット割込みの検出に伴う処理を実行します。

備考 1 この API 関数は、DALI 通信（コントロールデバイス）ストップビット割り込みに伴う処理 r_{Config_DALI}_interrupt_stop_bit_detection のコールバック・ルーチンとして呼び出されます。

備考 2 コールバック関数で必要なフラグの設定／クリアのみを保持し、他の処理コードは、コールバックおよび割り込み関数から移動する必要があります。 それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

```
static void    r_{Config_DALI}_callback_receiveend(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_DALI}_callback_error

受信エラーの検出に伴う処理を実行します。

備考 1　この API 関数は、DALI 通信（コントロールデバイス）受信エラー割り込みに伴う処理 r_{Config_DALI}_interrupt_error のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアのみを保持し、他の処理コードは、コールバックおよび割り込み関数から移動する必要があります。 それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

```
static void    r_{Config_DALI}_callback_error(uint16_t err_type);
```

[引数]

I/O	引数	説明
I	uint16_t err_type;	エラー種別を示す値 Bit 0：マンチェスタフレーミング・エラー Bit 1：オーバラン・エラー Bit 2：フレームサイズ違反・エラー Bit 3：ビットタイミング違反・エラー Bit 4 ~ 7：0

[戻り値]

なし

使用例

DALI 通信（コントロールデバイス）が 16 ビットのフレームデータを送信し、DALI 通信（コントロールギア）がデータを受信する例です。

main.c - DALI 通信（コントロールデバイス）

```
#include "r_smc_entry.h"

uint16_t tx_buf0[] = {0xFF66};    // 1 つの 16 ビット DALI フレーム（0xFF66）を含む送信バッファ。
volatile uint8_t sendend_flag = 0U;
void main(void);

void main(void)
{
    EI();    // グローバル割り込みを有効にして、システムが割り込みイベントに応答できるようにします。
    R_Config_DALI_Start();    // DALI 通信をコントロールデバイス（マスタ）として開始します。
    R_Config_DALI_Send(tx_buf0);    // 設定されたデバイスを使用して 16 ビット DALI フレームを送信します。
    while(1U != sendend_flag);    // 送信が完了するまで待ちます。
    R_Config_DALI_Stop();    // DALI 通信を停止します。
}
```

main.c - DALI 通信（コントロールギア）

```
#include "r_smc_entry.h"

uint8_t rx_buf0[];    // 受信したフレームを保持するポインタです。
uint8_t rx_buf1[100];    // 複数の受信フレームを格納するバッファです。
uint8_t p_rx_num = 0U;    // 現在のフレームで受信したバイト数へのポインタです。
uint8_t rx_num = 0U;    // 受信フレーム数の合計カウンタです。
volatile uint8_t receiveend_flag = 0U;
void main(void);

void main(void)
{
    EI();    // グローバル割り込みを有効にして、システムが割り込みイベントに応答できるようにします。

    R_Config_DALI_Start();    // DALI 通信をコントロールギア（スレーブ）として開始します。
    while(1U != receiveend_flag);    // 受信が完了するまで待ちます。
    // 受信したフレームを取得し、受信に成功した場合は格納します。
    if(R_Config_DALI1_GetReceivedFrame(&rx_buf0,&p_rx_num) == MD_OK)
    {
        rx_buf1[rx_num]= rx_buf0;    // 受信したバイト/フレームをバッファに格納します。
        rx_num++;    // 受信フレームのカウンタをインクリメントします。
    }

    R_Config_ DALI1_Stop();    // DALI 通信を停止します。
}
```

Config_DALI_Device_user.c

```
/* Start user code for global. Do not edit comment generated here */
extern volatile uint8_t sendend_flag;
/* End user code. Do not edit comment generated here */

static void r_Config_DALI_Device_callback_sendend (void)
{
    /* Start user code for r_Config_DALI_Device_callback_sendend. Do not edit comment generated here */
    sendend_flag++;
    /* End user code. Do not edit comment generated here */
}
```

Config_DALI1_Gear_user.c

```
/* Start user code for global. Do not edit comment generated here */
extern volatile uint8_t receiveend_flag;
/* End user code. Do not edit comment generated here */

static void r_Config_DALI_Gear_callback_receiveend (void)
{
    /* Start user code for r_Config_DALI_Gear_callback_receiveend. Do not edit comment generated here */
    receiveend_flag++;
    /* End user code. Do not edit comment generated here */
}
```

4.2.53 DALI 通信（コントロールギア）

以下に、スマート・コンフィグレータが DALI 通信（コントロールギア）モジュール用として出力する API 関数の一覧を示します。

表 4.59 DALI 通信（コントロールギア）API 関数

API 関数名	周辺機能	機能概要
R_{Config_DALI}_Create	DALI	DALI 通信（コントロールギア）モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_DALI}_Start		DALI 通信（コントロールギア）モジュールの動作を開始します。
R_{Config_DALI}_Stop		DALI 通信（コントロールギア）モジュールの動作を終了します。
R_{Config_DALI}_SoftwareReset		DALI 通信（コントロールギア）モジュールをリセットします。
R_{Config_DALI}_EnableForceActiveState		DALITxD0 アサートを有効にし、アサートレベル (アクティブ状態) が Low であることを示します。 DALITxD0 端子からの出力を Low にします。
R_{Config_DALI}_DisableForceActiveState		DALITxD0 アサートを無効にします。内部送信データを DALITxD0 端子から出力します。
R_{Config_DALI}_GetStatus		DALI 通信（コントロールギア）の状態を取得します。
R_{Config_DALI}_Send		フレームデータを送信します。フレーム長は GUI で設定され、固定値になります。
R_{Config_DALI}_GetReceivedFrame		フレームデータとフレーム長を受信します。
R_{Config_DALI}_Create_UserInit		DALI 通信（コントロールギア）に関するユーザ独自の初期化処理を実行します。
r_{Config_DALI}_interrupt_error		DALI 通信（コントロールギア）の受信エラー割り込み（INTED）に伴う処理を実行します。
r_{Config_DALI}_interrupt_falling_edge_detection		DALI 通信（コントロールギア）の立ち下がりエッジ割り込み（INTFED）に伴う処理を実行します。
r_{Config_DALI}_interrupt_power_down_detection		DALI 通信（コントロールギア）のパワーダウン割り込み（INTBPD）に伴う処理を実行します。
r_{Config_DALI}_interrupt_stop_bit_detection		DALI 通信（コントロールギア）のストップビット検出割り込み（INTSDD）に伴う処理を実行します。
r_{Config_DALI}_callback_sendend		送信完了の検出に伴う処理を実行します。
r_{Config_DALI}_callback_receiveend		受信完了の検出に伴う処理を実行します。
r_{Config_DALI}_callback_error		受信エラーの検出に伴う処理を実行します。

R_{Config_DALI}_Create

DALI 通信（コントロールギア）モジュールを制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_DALI}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_DALI}_Start

DALI 通信（コントロールギア）モジュールの動作を開始します。

[指定形式]

```
void    R_{Config_DALI}_Start(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_DALI}_Stop

DALI 通信（コントロールギア）モジュールの動作を終了します。

[指定形式]

```
void    R_{Config_DALI}_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_DALI}_SoftwareReset

DALI 通信（コントロールギア）モジュールをリセットします。

[指定形式]

```
void    R_{Config_DALI}_SoftwareReset(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_DALI}_EnableForceActiveState

DALITxD0 アサートを有効にし、アサートレベル (アクティブ状態) が Low であることを示します。DALITxD0 端子からの出力を Low にします。

[指定形式]

```
void    R_{Config_DALI}_ EnableForceActiveState (void);
```

[引数]

なし

[戻り値]

なし

R_{Config_DALI}_DisableForceActiveState

DALITxD0 アサートを無効にします。内部送信データを DALITxD0 端子から出力します。

[指定形式]

```
void    R_{Config_DALI}_ DisableForceActiveState (void);
```

[引数]

なし

[戻り値]

なし

R_{Config_DALI}_GetStatus

DALI 通信（コントロールギア）の状態を取得します。

[指定形式]

```
void    R_{Config_DALI}_GetStatus(uint16_t * const status);
```

[引数]

I/O	引数	説明
O	uint16_t * const status;	DALI 状態レジスタバッファへのポインタ

[戻り値]

なし

R_{Config_DALI}_Send

フレームデータを送信します。フレーム長は 8 ビット固定値になります。

備考　この API 関数は、引数 tx_buf で指定されたバッファのデータをレジスタ TDR1L に設定します。 そして、レジスタ TDR1L の送信が完了後、r_{Config_DALI}_interrupt_stop_bit_detection に入ります。

[指定形式]

```
void    R_{Config_DALI}_Send(uint8_t tx_buf);
```

[引数]

I/O	引数	説明
I	uint8_t tx_buf;	送信バッファ

[戻り値]

なし

R_{Config_DALI}_GetReceivedFrame

フレームデータとフレーム長を受信します。

[指定形式]

```
MD_STATUS R_{Config_DALI}_GetReceivedFrame(uint32_t * const rx_buf, uint16_t * const rx_num);
```

[引数]

I/O	引数	説明
O	Uint32_t * const rx_buf;	受信バッファへのポインタ
I	uint16_t tx_num;	バッファフレーム長

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_ARGERROR	引数の指定が不正

R_{Config_DALI}_Create_UserInit

DALI 通信（コントロールギア）に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_DALI}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_DALI}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_DALI}_interrupt_error

DALI 通信（コントロールギア）の受信エラー割り込み（INTED）に伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_DALI}_interrupt_error(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_DALI}_interrupt_error(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_DALI}_interrupt_error(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_DALI}_interrupt_falling_edge_detection

DALI 通信（コントロールギア）の立ち下がりエッジ割り込み（INTFED）に伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_DALI}_interrupt_falling_edge_detection(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_DALI}_interrupt_falling_edge_detection(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_DALI}_interrupt_falling_edge_detection(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_DALI}_interrupt_power_down_detection

DALI 通信（コントロールギア）のパワーダウン割り込み（INTBPD）に伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_DALI}_interrupt_power_down_detection(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_DALI}_interrupt_power_down_detection(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_DALI}_interrupt_power_down_detection(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_DALI}_interrupt_stop_bit_detection

DALI 通信（コントロールギア）のストップビット検出割り込み（INTSDD）に伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_DALI}_interrupt_stop_bit_detection(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_DALI}_interrupt_stop_bit_detection(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_DALI}_interrupt_stop_bit_detection(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_DALI}_callback_sendend

ストップビット割込みの検出に伴う処理を実行します。

備考 1　この API 関数は、DALI 通信（コントロールギア）ストップビット割り込みに伴う処理 r_{Config_DALI}_interrupt_stop_bit_detection のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアのみを保持し、他の処理コードは、コールバックおよび割り込み関数から移動する必要があります。 それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

```
static void    r_{Config_DALI}_callback_sendend(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_DALI}_callback_receiveend

ストップビット割り込みの検出に伴う処理を実行します。

備考 1　この API 関数は、DALI 通信（コントロールギア）ストップビット割り込みに伴う処理 r_{Config_DALI}_interrupt_stop_bit_detection のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアのみを保持し、他の処理コードは、コールバックおよび割り込み関数から移動する必要があります。 それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

```
static void    r_{Config_DALI}_callback_receiveend(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_DALI}_callback_error

受信エラーの検出に伴う処理を実行します。

備考 1　この API 関数は、DALI 通信（コントロールギア）受信エラー割り込みに伴う処理 r_{Config_DALI}_interrupt_error のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアのみを保持し、他の処理コードは、コールバックおよび割り込み関数から移動する必要があります。 それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

```
static void    r_{Config_DALI}_callback_error(uint16_t err_type);
```

[引数]

I/O	引数	説明
I	uint16_t err_type;	エラー種別を示す値 Bit 0：マンチェスタフレーミング・エラー Bit 1：オーバラン・エラー Bit 2：フレームサイズ違反・エラー Bit 3：ビットタイミング違反・エラー Bit 4 ~ 7：0

[戻り値]

なし

使用例

DALI 通信（コントロールデバイス）の使用例を参照してください。

4.2.54 IIC 通信 (マスタ・モード) (シリアル・アレイ・ユニット)

以下に、スマート・コンフィグレータが IIC 通信 (マスタ・モード) (シリアル・アレイ・ユニット) 用として出力する API 関数の一覧を示します。

表 4.60 IIC 通信 (マスタ・モード) (シリアル・アレイ・ユニット) 用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_IIC*r*}_Create	シリアル・アレイ・ユニット	IIC*r* マスタ・モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_IIC*r*}_StartCondition		IIC*r* スタート・コンディションを発行します。
R_{Config_IIC*r*}_StopCondition		IIC*r* ストップ・コンディションを発行します。
R_{Config_IIC*r*}_Stop		IIC*r* モジュールを停止します。
R_{Config_IIC*r*}_Master_Send		マスタ・モードで IIC*r* データ送信を開始します。
R_{Config_IIC*r*}_Master_Receive		マスタ・モードで IIC*r* データ受信を開始します。
R_{Config_IIC*r*}_Create_UserInit		IIC*r* に関するユーザ独自の初期化処理を実行します。
r_{Config_IIC*r*}_interrupt		INTIIC*r* 転送完了割り込みに伴う処理を実行します。
r_{Config_IIC*r*}_callback_master_sendend		IIC*r* マスタ送信完了の検出に伴う処理を実行します。
r_{Config_IIC*r*}_callback_master_receiveend		IIC*r* マスタ受信完了の検出に伴う処理を実行します。
r_{Config_IIC*r*}_callback_master_error		IIC*r* マスタ転送エラーの検出に伴う処理を実行します。

R_{Config_IIC*r*}_Create

IIC*r* マスタ・モードを制御する前に必要な初期化処理を実行します。

備考 1. この API 関数は、R_SAU*m*_Create から呼び出されます。

備考 2. *m*＝0 のとき *r*＝00, 01, 10, 11
m＝1 のとき *r*＝20, 21, 30, 31

[指定形式]

```
void    R_{Config_IIC r}_Create(void);
```

備考 *r*＝00, 01, 10, 11, 20, 21, 30, 31

[引数]

なし

[戻り値]

なし

R_{Config_IICr}_StartCondition

IIC*r* スタート・コンディションを発行します。

備考　この API 関数は、R_{Config_IICr}_StartCondition および R_{Config_IICr}_StopCondition の内部関数として使用されます。

[指定形式]

```
void    R_{Config_IICr}_StartCondition(void);
```

備考　*r*＝00, 01, 10, 11, 20, 21, 30, 31

[引数]

なし

[戻り値]

なし

R_{Config_IIC*r*}_StopCondition

IIC*r* ストップ・コンディションを発行します。

[指定形式]

```
void    R_{Config_IIC r}_StopCondition(void);
```

備考　　*r*＝00, 01, 10, 11, 20, 21, 30, 31

[引数]

なし

[戻り値]

なし

R_{Config_IIC*r*}_Stop

IIC*r* モジュールを停止します。

[指定形式]

```
void    R_{Config_IIC r}_Stop(void);
```

備考　　*r*＝00, 01, 10, 11, 20, 21, 30, 31

[引数]

なし

[戻り値]

なし

R_{Config_IIC*r*}_Master_Send

マスタ・モードで IIC*r* データ送信を開始します。

備考 1. この API 関数は、引数 tx_buf で指定されたバッファから 1 バイト単位の IICA マスタ送信を、引数 tx_num で指定された回数分繰り返します。

備考 2. この API を呼び出す前に、通信が停止／一時停止され、SDA／SCL がハイ・レベルであることを確認してください。

[指定形式]

```
void    R_{Config_IIC r}_Master_Send(uint8_t adr, uint8_t * const tx_buf, uint16_t tx_num);
```

備考 *r*＝00, 01, 10, 11, 20, 21, 30, 31

[引数]

I/O	引数	説明
I	uint8_t adr;	スレーブ・アドレス
I	uint8_t * const tx_buf;	送信するデータを格納したバッファへのポインタ
I	uint16_t tx_num;	送信するデータの総数

[戻り値]

なし

R_{Config_IIC*r*}_Master_Receive

マスタ・モードで IIC*r* データ受信を開始します。

備考 1. この API 関数は、1 バイト単位の IICA マスタ受信を引数 rx_num で指定された回数分繰り返し、引数 rx_buf で指定されたバッファに格納します。

備考 2. この API を呼び出す前に、通信が停止／一時停止され、SDA／SCL がハイ・レベルであることを確認してください。

[指定形式]

```
void    R_{Config_IIC*r*}_Master_Receive(uint8_t adr, uint8_t * const rx_buf, uint16_t rx_num);
```

備考 *r*＝00, 01, 10, 11, 20, 21, 30, 31

[引数]

I/O	引数	説明
I	uint8_t adr;	スレーブ・アドレス
O	uint8_t * const rx_buf;	受信したデータを格納するバッファへのポインタ
O	uint16_t rx_num;	受信するデータの総数

[戻り値]

なし

R_{Config_IIC*r*}_Create_UserInit

IIC*r* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_IIC*r*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_IICr}_Create_UserInit(void);
```

備考　*r*＝00, 01, 10, 11, 20, 21, 30, 31

[引数]

なし

[戻り値]

なし

r_{Config_IIC*r*}_interrupt

INTIIC*r* 転送完了割り込みに伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_IIC r}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_IIC r}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_IIC r}_interrupt(void);
```

備考　　*r*＝00, 01, 10, 11, 20, 21, 30, 31

[引数]

なし

[戻り値]

なし

r_{Config_IIC*r*}_callback_master_sendend

IIC*r* マスタ送信完了の検出に伴う処理を実行します。

備考 1　この API 関数は、INTIIC*r* 転送完了割り込み処理 r_{Config_IIC*r*}_interrupt のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアを維持し、他の処理コードをコールバックおよび割り込み関数から移動することに注意してください。それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_IIC*r*}_callback_master_sendend(void);

備考　*r*＝00, 01, 10, 11, 20, 21, 30, 31

[引数]

なし

[戻り値]

なし

r_{Config_IIC*r*}_callback_master_receiveend

IICr マスタ受信完了の検出に伴う処理を実行します。

備考 1　この API 関数は、INTIIC*r* 転送完了割り込み処理 r_{Config_IIC*r*}_interrupt のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアを維持し、他の処理コードをコールバックおよび割り込み関数から移動することに注意してください。それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_IIC*r*}_callback_master_receiveend(void);

備考　*r*＝00, 01, 10, 11, 20, 21, 30, 31

[引数]

なし

[戻り値]

なし

r_{Config_IIC*r*}_callback_master_error

IICr マスタ転送エラーの検出に伴う処理を実行します。

備考 1　この API 関数は、INTIIC*r* 転送完了割り込み処理 r_{Config_IIC*r*}_interrupt のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアを維持し、他の処理コードをコールバックおよび割り込み関数から移動することに注意してください。それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_IIC*r*}_callback_error(MD_STATUS flag);

備考　*r*＝00, 01, 10, 11, 20, 21, 30, 31

[引数]

I/O	引数	説明
I	MD_STATUS flag;	ステータス・フラグ MD_NACK：NACK エラー MD_OVERRUN：オーバラン・エラー

[戻り値]

なし

使用例

IIC0 マスタと IICA0 スレーブの通信例です（IIC0 マスタ送信およびマスタ受信モードを含む）。

main.c

```
#include "r_smc_entry.h"

uint8_t tx_buf[6] = {0xA5,0x3F,0xC0,0x5C,0xB6,0x37};    // 6 バイトのデータを含む送信バッファ。
uint8_t rx_buf1[6] = {0x00,0x00,0x00,0x00,0x00,0x00};    // 0 に初期化した受信バッファ 1。
uint8_t rx_buf2[6] = {0x00,0x00,0x00,0x00,0x00,0x00};    // 0 に初期化した受信バッファ 2。
volatile uint8_t transmitend_flag = 0U;
volatile uint8_t receiveend_flag = 0U;

void main(void);

void main(void)
{
    EI();    // グローバル割り込みを有効にして、システムが割り込みイベントに応答できるようにします。
    R_Config_IIC00_StartCondition();    // IIC スタート・コンディションを開始します。

    R_Config_IIC00_Master_Receive(0x24,rx_buf1,sizeof(rx1_buf));    // マスタはスレーブ・アドレス 0x00 から rx_buf1 にデータを受信します。
    R_Config_IICA1_Slave_Send(tx_buf,sizeof(tx_buf));    // スレーブは rx_buf1 からデータを送信します。

    while(receiveend_flag != 1);    // 受信が完了するまで待ちます。
    transmitend = 0;    // 送信完了フラグをリセットします。
    receiveend = 0;    // 受信完了フラグをリセットします。

    R_Config_IIC00_StopCondition();    // ストップ・コンディションを送信して、IIC 通信を終了します。
    R_Config_IIC00_StartCondition();    // 新しい IIC 通信を開始します。

    R_Config_IICA1_Slave_Receive(rx_buf2, sizeof(rx_buf2));    // スレーブはデータを rx_buf2 に受信します。
    R_Config_IIC00_Master_Send(0x24, tx_buf,sizeof(tx_buf));    // マスタは tx_buf からスレーブ・アドレス 0x24 にデータを送信します。

    while(receiveend_flag != 1);    // 受信が完了するまで待ちます。
    transmitend_flag = 0;    // 送信完了フラグをリセットします。
    receiveend_flag = 0;    // 受信完了フラグをリセットします。

    R_Config_IIC00_StopCondition();    // ストップ・コンディションを送信して、IIC 通信を終了します。
    R_Config_IIC00_Stop();    // IIC00 モジュールを停止します。
    R_Config_IICA0_Stop();    // IICA0 モジュールを停止します。
}
```

Config_IIC00_user.c

```
/* Start user code for global. Do not edit comment generated here */
extern volatile uint8_t transmitend_flag;
extern uint8_t receiveend_flag;
/* End user code. Do not edit comment generated here */

static void r_Config_IIC00_callback_master_sendend (void)
{
   /* Start user code for r_Config_IIC00_callback_master_sendend. Do not edit comment generated here */
   transmitend_flag = 1U;   // 送信完了を示すフラグを設定します。
   /* End user code. Do not edit comment generated here */
}

static void r_Config_IIC00_callback_master_receiveend (void)
{
   /* Start user code for r_Config_IIC00_callback_master_receiveend. Do not edit comment generated here */
   receiveend_flag = 1U;   // 受信完了を示すフラグを設定します。
   /* End user code. Do not edit comment generated here */
}
```

Config_IICA0_user.c

```
/* Start user code for global. Do not edit comment generated here */
extern volatile uint8_t transmitend_flag;
extern uint8_t receiveend_flag;
/* End user code. Do not edit comment generated here */

static void r_Config_IICA0_callback_slave_sendend(void)
{
   /* Start user code for r_Config_IICA0_callback_slave_sendend. Do not edit comment generated here */
   transmitend_flag = 1U;   // スレーブ送信完了を示すフラグを設定します。
   /* End user code. Do not edit comment generated here */
}

static void r_Config_IICA0_callback_slave_receiveend(void)
{
   /* Start user code for r_Config_IICA0_callback_slave_receiveend. Do not edit comment generated here */
   receiveend_flag = 1U;   // スレーブ受信完了を示すフラグを設定します。
   /* End user code. Do not edit comment generated here */
}
```

4.2.55 IIC 通信 (マスタ・モード) (シリアル・インタフェース IICA)

以下に、スマート・コンフィグレータが IIC 通信 (マスタ・モード) (シリアル・インタフェース IICA) 用として出力する API 関数の一覧を示します。

表 4.61 IIC 通信 (マスタ・モード) (シリアル・インタフェース IICA) 用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_IICA*n*}_Create	シリアル・インタフェース IICA	IICA*n* マスタ・モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_IICA*n*}_StopCondition		IICA*n* ストップ・コンディションを発行します。
R_{Config_IICA*n*}_Stop		IICA*n* マスタの動作を停止します。
R_{Config_IICA*n*}_Master_Send		マスタ・モードでデータ送信を開始します。
R_{Config_IICA*n*}_Master_Receive		マスタ・モードでデータ受信を開始します。
R_{Config_IICA*n*}_Check_Comstate		通信状況を読み出します。
R_{Config_IICA*n*}_Poll		通信状況を確認します。ステータス変数の値で判定します。
R_{Config_IICA*n*} _Wait_Comend		通信が完了するまで関数内で待機します。
R_{Config_IICA*n*}_Bus_Check		バスのステータスを確認し解放されている場合は、スタート・コンディションを発行します。
R_{Config_IICA*n*}_StartCondition		スタート・コンディションを発行します。
R_{Config_IICA*n*}_Wait_Time		50us 待機します。
R_{Config_IICA*n*}_Create_UserInit		IICA*n* に関するユーザ独自の初期化処理を実行します。
r_{Config_IICA*n*}_interrupt		IICA*n* 通信割り込み（INTIICA*n*）に伴う処理を実行します。
r_{Config_IICA*n*}_master_handler		マスタ・モードでの IICA*n* データの送受信およびエラーを制御します。
r_{Config_IICA*n*}_callback_master_senddend		マスタ送信完了の検出に伴う処理を実行します。
r_{Config_IICA*n*}_callback_master_receiveend		マスタ受信完了の検出に伴う処理を実行します。
r_{Config_IICA*n*}_callback_master_error		マスタ・モードでのエラー検出に伴う処理を実行します。

R_{Config_IICA*n*}_Create

IICA*n* マスタ・モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_IICAn}_Create(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_IICA*n*}_StopCondition

IICA*n* ストップ・コンディションを発行します。

[指定形式]

```
uint8_t    R_{Config_IICAn}_StopCondition(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

マクロ	説明
SUCCESS	動作完了
BUS_FREE	バス解放状態（成功）
BUS_ERROR	バスまたは IICA*n* がビジー状態

R_{Config_IICA*n*}_Stop

IICA*n* マスタの動作を停止します。

[指定形式]

```
void    R_{Config_IICAn}_Stop(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_IICA*n*}_Master_Send

マスタ・モードでデータ送信を開始します。

[指定形式]

```
void R_{Config_IICAn}_Master_Send(uint8_t sladr8, uint8_t * const tx_buf, uint16_t tx_num, uint8_t wait);
```

備考 *n* はチャネル番号を示します。

[引数]

I/O	引数	説明
I	uint8_t sladr8;	送信アドレス
I	uint8_t * const tx_buf;	送信するデータを格納したバッファへのポインタ
I	uint16_t tx_num;	送信するバイト数
I	uint8_t wait;	スタート・コンディション待ち

[戻り値]

マクロ	説明
BUS_ERROR	バスまたは IICA*n* がビジー状態
SUCCESS	動作完了

R_{Config_IICA*n*}_Master_Receive

マスタ・モードでデータ受信を開始します。

[指定形式]

```
void    R_{Config_IICAn}_Master_Receive(uint8_t sladr8, uint8_t * const rx_buf, uint16_t rx_num,
uint8_t wait);
```

備考　　*n* はチャネル番号を示します。

[引数]

I/O	引数	説明
I	uint8_t sladr8;	受信アドレス
O	uint8_t * const rx_buf;	受信したデータを格納したバッファへのポインタ
O	uint16_t rx_num;	受信するバイト数
	uint8_t wait	スタート・コンディション待ち

[戻り値]

マクロ	説明
COM_ERROR	その他の通信エラー状態
SUCCESS	動作完了

R_{Config_IICA*n*}_Check_Comstate

通信状況を読み出します。

[指定形式]

```
uint8_t    R_{Config_IICAn}_Check_Comstate(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

マクロ	説明
g_iica*n*_status	-

R_{Config_IICA*n*}_Poll

通信状況を確認します。ステータス変数の値で判定します。

[指定形式]

```
uint8_t   R_{Config_IICAn}_Poll(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

マクロ	説明
SUCCESS	動作完了
ON_COMMU	通信状態
BUS_ERROR	バスまたは IICA*n* がビジー状態
NO_SLAVE	スレーブ・アドレスに対する NACK
NO_ACK	送信データに対する NACK

R_{Config_IICA*n*}_Wait_Comend

通信が完了するまで関数内で待機します。

[指定形式]

```
uint8_t    R_{Config_IICAn}_Wait_Comend(uint8_t stop);
```

備考　　*n* はチャネル番号を示します。

[引数]

I/O	引数	説明
I	uint8_t stop;	ストップ・コンディションの発行

[戻り値]

マクロ	説明
ON_COMMU	通信状態
SUCCESS	動作完了

R_{Config_IICA*n*}_Bus_Check

バスのステータスを確認し解放されている場合は、スタート・コンディションを発行します。

[指定形式]

```
uint8_t    R_{Config_IICAn}_Bus_Check(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

マクロ	説明
SUCCESS	動作完了
BUS_HOLD	IIC バスをホールド状態（同上）
BUS_ERROR	バスまたは IICA*n* がビジー状態

R_{Config_IICA*n*}_StartCondition

スタート・コンディションを発行します。

[指定形式]

```
uint8_t    R_{Config_IICAn}_StartCondition(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

マクロ	説明
SUCCESS	動作完了
BUS_ERROR	バスまたは IICA*n* がビジー状態

R_{Config_IICA*n*}_Wait_Time

50us 待機します。

[指定形式]

```
void    R_{Config_IICAn}_Wait_Time(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_IICA*n*}_Create_UserInit

IICA*n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_IICA*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_IICAn}_Create_UserInit(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_IICA*n*}_interrupt

IICA*n* 通信割り込み（INTIICA*n*）に伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_IICAn}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_IICAn}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_IICAn}_interrupt(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_IICA*n*}_master_handler

マスタ・モードでの IICA*n* データの送受信およびエラーを制御します。

備考　この API 関数は、IICA*n* 通信割り込みに伴う処理 r_{Config_ IICA*n*}_interrupt のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_IICAn}_master_handler(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_IICA*n*}_callback_master_sendend

マスタ送信完了の検出に伴う処理を実行します。

備考 1　この API 関数は、割り込み処理 r_{Config_IICA*n*}_master_handler のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアを維持し、他の処理コードをコールバックおよび割り込み関数から移動することに注意してください。それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_IICA*n*}_callback_master_sendend(void);

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_IICA*n*}_callback_master_receiveend

マスタ受信完了の検出に伴う処理を実行します。

備考 1　この API 関数は、割り込み処理 r_{Config_IICA*n*}_master_handler のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアを維持し、他の処理コードをコールバックおよび割り込み関数から移動することに注意してください。それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_IICA*n*}_callback_master_receiveend(void);

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_IICA*n*}_callback_master_error

マスタ・モードでのエラー検出に伴う処理を実行します。

備考 1　この API 関数は、割り込み処理 r_{Config_IICA*n*}_master_handler のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアを維持し、他の処理コードをコールバックおよび割り込み関数から移動することに注意してください。それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_IICA*n*}_callback_master_error(MD_STATUS flag);

備考　*n* はチャネル番号を示します。

[引数]

I/O	引数	説明
I	MD_STATUS flag;	ステータス・フラグ MD_NACK：NACK エラー

[戻り値]

なし

使用例

IICA0 マスタと ICA1 スレーブの通信例です（送信モードと受信モードの両方を含む）。

main.c (1/2)

```
#include "r_smc_entry.h"

uint8_t tx_buf[6] = {0xA5,0x3F,0xC0,0x5C,0xB6,0x37};    // 6 バイトのデータを含む送信バッファ。
uint8_t rx_buf1[6] = {0x00,0x00,0x00,0x00,0x00,0x00};    // 0 に初期化した受信バッファ 1。
uint8_t rx_buf2[6] = {0x00,0x00,0x00,0x00,0x00,0x00};    // 0 に初期化した受信バッファ 2。
volatile uint8_t transmitend_flag = 0U;
volatile uint8_t receiveend_flag = 0U;

void main(void);

void main(void)
{
    EI();    // グローバル割り込みを有効にして、システムが割り込みイベントに応答できるようにします。

    /* Check bus status */
    if (R_Config_IICA0_Bus_Check() != 0U)
    {
        // バスがビジーで、待機またはエラーを処理します。
        R_Config_IICA0_Wait_Time();
    }

    /* Start condition */
    if (R_Config_IICA0_StartCondition() == 0U)
    {
        // 正常に開始。
    }

    /* Slave ready to receive */
    R_Config_IICA1_Slave_Receive(rx_buf1, sizeof(tx_buf));     // スレーブは rx_buf1 にデータを受信します。

    /* Master sends data */
    if (R_Config_IICA0_Master_Send(0x24, tx_buf, sizeof(tx_buf),100) == 0U)     // マスタはアドレス 0x24 のスレーブにデータを送信します。
    {
        // 送信開始に成功。
    }

    /* Wait for communication to finish */
    while (R_Config_IICA0_Check_Comstate() != 0U)
    {
        R_Config_IICA0_Poll(); // 通信ステータスをポーリング。
    }
    R_Config_IICA0_Wait_Comend(1U); // 通信完了を待って STOP を送信します。

    while(receiveend_flag != 1);    // 受信が完了するまで待ちます。
    transmitend_flag = 0;    // 送信完了フラグをリセットします。
    receiveend_flag = 0;    // 受信完了フラグをリセットします。
```

main.c (2/2)

```
    /* Start condition again */
    R_Config_IICA0_StartCondition();

    /* Master receives data */
    if (R_Config_IICA0_Master_Receive(0x24, rx_buf2, sizeof(rx_buf2),100) == 0U)    // マスタはスレ
ーブからデータを受信します。
    {
        // 受信開始に成功。
    }

    /* Wait for communication to finish */
    while (R_Config_IICA0_Check_Comstate() != 0U)
    {
        R_Config_IICA0_Poll();
    }
    R_Config_IICA0_Wait_Comend(1U);

    while (receiveend_flag != 1U);
    transmitend_flag = 0U;
    receiveend_flag = 0U;

    /* Slave sends data */
    R_Config_IICA1_Slave_Send(tx_buf, sizeof(tx_buf));

    while (receiveend_flag != 1U);
    transmitend_flag = 0U;
    receiveend_flag = 0U;

    /* Stop communication */
    R_Config_IICA0_Stop();    // IICA0 通信を停止します。
    R_Config_IICA1_Stop();    // IICA1 通信を停止します。
}
```

Config_IICA0_user.c

```
/* Start user code for global. Do not edit comment generated here */
extern volatile uint8_t transmitend_flag;
extern volatile uint8_t receiveend_flag;
/* End user code. Do not edit comment generated here */

static void r_Config_IICA0_callback_master_sendend (void)
{
   SPT0 = 1U;
   /* Start user code for r_Config_IICA0_callback_master_sendend. Do not edit comment generated here */
   transmitend_flag = 1U;   // 送信完了を示すフラグを設定します。
   /* End user code. Do not edit comment generated here */
}

static void r_Config_IICA0_callback_master_receiveend (void)
{
   SPT0 = 0U;
   /* Start user code for r_Config_IICA0_callback_master_receiveend. Do not edit comment generated here */
   receiveend_flag = 1U;   // 受信完了を示すフラグを設定します。
   /* End user code. Do not edit comment generated here */
}
```

Config_IICA1_user.c

```
/* Start user code for global. Do not edit comment generated here */
extern volatile uint8_t transmitend_flag;
extern volatile uint8_t receiveend_flag;
/* End user code. Do not edit comment generated here */

static void r_Config_IICA1_callback_slave_sendend(void)
{
   /* Start user code for r_Config_IICA1_callback_slave_sendend. Do not edit comment generated here */
   transmitend_flag = 1U;   // スレーブ送信完了を示すフラグを設定します。
   /* End user code. Do not edit comment generated here */
}

static void r_Config_IICA1_callback_slave_receiveend(void)
{
   /* Start user code for r_Config_IICA1_callback_slave_receiveend. Do not edit comment generated here */
   receiveend_flag = 1U;   // スレーブ受信完了を示すフラグを設定します。
   /* End user code. Do not edit comment generated here */
}
```

4.2.56 IIC 通信 (マスタ・モード、EEPROM 通信) (シリアル・インタフェース IICA)

以下に、スマート・コンフィグレータが IIC 通信 (マスタ・モード、EEPROM 通信) (シリアル・インタフェース IICA) 用として出力する API 関数の一覧を示します。

表 4.62 IIC 通信 (マスタ・モード、EEPROM 通信) (シリアル・インタフェース IICA) 用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_IICA*n*}_Create	シリアル・インタフェース IICA	IICA*n* マスタ・モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_IICA*n*}_StopCondition		IICA*n* ストップ・コンディションを発行します。
R_{Config_IICA*n*}_Stop		IICA*n* マスタの動作を停止します。
R_{Config_IICA*n*}_Master_Send		マスタ・モードでデータ送信を開始します。
R_{Config_IICA*n*}_Master_Receive		マスタ・モードでデータ受信を開始します。
R_{Config_IICA*n*}_Check_Comstate		通信状況を読み出します。
R_{Config_IICA*n*}_Poll		通信状況を確認します。ステータス変数の値で判定します。
R_{Config_IICA*n*} _Wait_Comend		通信が完了するまで関数内で待機します。
R_{Config_IICA*n*}_Bus_Check		バスのステータスを確認し解放されている場合は、スタート・コンディションを発行します。
R_{Config_IICA*n*}_StartCondition		スタート・コンディションを発行します。
R_{Config_IICA*n*}_Wait_Time		50us 待機します。
R_{Config_IICA*n*}_Create_UserInit		IICA*n* に関するユーザ独自の初期化処理を実行します。
r_{Config_IICA*n*}_interrupt		IICA*n* 通信割り込み（INTIICA*n*）に伴う処理を実行します。
r_{Config_IICA*n*}_master_handler		マスタ・モードでの IICA*n* データの送受信およびエラーを制御します。
r_{Config_IICA*n*}_callback_master_senddend		マスタ送信完了の検出に伴う処理を実行します。
r_{Config_IICA*n*}_callback_master_receiveend		マスタ受信完了の検出に伴う処理を実行します。
r_{Config_IICA*n*}_callback_master_error		マスタ・モードでのエラー検出に伴う処理を実行します。

R_{Config_IICA*n*}_Create

IICA*n* マスタ・モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_IICAn}_Create(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_IICA*n*}_StopCondition

IICA*n* ストップ・コンディションを発行します。

[指定形式]

```
uint8_t    R_{Config_IICAn}_StopCondition(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

マクロ	説明
SUCCESS	動作完了
BUS_FREE	バス解放状態（成功）
BUS_ERROR	バスまたは IICA*n* がビジー状態

R_{Config_IICA*n*}_Stop

IICA*n* マスタの動作を停止します。

[指定形式]

```
void    R_{Config_IICAn}_Stop(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_IICA*n*}_Master_Send

マスタ・モードでデータ送信を開始します。

[指定形式]

```
void    R_{Config_IICAn}_Master_Send(uint8_t sladr7, uint8_t adr, uint8_t * const tx_buf, uint16_t tx_num, uint8_t wait);
```

備考　　*n* はチャネル番号を示します。

[引数]

I/O	引数	説明
I	uint8_t sladr7;	スレーブ・アドレス
I	uint8_t adr;	送信アドレス
I	uint8_t * const tx_buf;	送信するデータを格納したバッファへのポインタ
I	uint16_t tx_num;	送信するバイト数
I	uint8_t wait;	スタート・コンディション待ち

[戻り値]

マクロ	説明
BUS_ERROR	バスまたは IICA*n* がビジー状態
SUCCESS	動作完了

R_{Config_IICA*n*}_Master_Receive

マスタ・モードでデータ受信を開始します。

[指定形式]

```
void R_{Config_IICAn}_Master_Receive(uint8_t sladr7, uint8_t adr, uint8_t * const rx_buf, uint16_t rx_num, uint8_t wait);
```

備考 *n* はチャネル番号を示します。

[引数]

I/O	引数	説明
I	uint8_t sladr7;	スレーブ・アドレス
I	uint8_t adr;	受信アドレス
O	uint8_t * const rx_buf;	受信したデータを格納したバッファへのポインタ
O	uint16_t rx_num;	受信するバイト数
	uint8_t wait	スタート・コンディション待ち

[戻り値]

マクロ	説明
COM_ERROR	その他の通信エラー状態
SUCCESS	動作完了

R_{Config_IICA*n*}_Check_Comstate

通信状況を読み出します。

[指定形式]

```
uint8_t    R_{Config_IICAn}_Check_Comstate(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

マクロ	説明
g_iica*n*_status	-

R_{Config_IICA*n*}_Poll

通信状況を確認します。ステータス変数の値で判定します。

[指定形式]

```
uint8_t    R_{Config_IICAn}_Poll(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

マクロ	説明
SUCCESS	動作完了
ON_COMMU	通信状態
BUS_ERROR	バスまたは IICA*n* がビジー状態
NO_SLAVE	スレーブ・アドレスに対する NACK
NO_ACK	送信データに対する NACK

R_{Config_IICA*n*}_Wait_Comend

通信が完了するまで関数内で待機します。

[指定形式]

```
uint8_t    R_{Config_IICAn}_Wait_Comend(uint8_t stop);
```

備考　　*n* はチャネル番号を示します。

[引数]

I/O	引数	説明
I	uint8_t stop;	ストップ・コンディションの発行

[戻り値]

マクロ	説明
ON_COMMU	通信状態
SUCCESS	動作完了

R_{Config_IICA*n*}_Bus_Check

バスのステータスを確認し解放されている場合は、スタート・コンディションを発行します。

[指定形式]

```
uint8_t    R_{Config_IICAn}_Bus_Check(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

マクロ	説明
SUCCESS	動作完了
BUS_HOLD	IIC バスをホールド状態（同上）
BUS_ERROR	バスまたは IICA*n* がビジー状態

R_{Config_IICA*n*}_StartCondition

スタート・コンディションを発行します。

[指定形式]

```
uint8_t    R_{Config_IICAn}_StartCondition(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

マクロ	説明
SUCCESS	動作完了
BUS_ERROR	バスまたは IICA*n* がビジー状態

R_{Config_IICA*n*}_Wait_Time

50us 待機します。

[指定形式]

void R_{Config_IICA*n*}_Wait_Time(void);

備考 *n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_IICA*n*}_Create_UserInit

IICA*n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_IICA*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_IICAn}_Create_UserInit(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_IICA*n*}_interrupt

IICA*n* 通信割り込み（INTIICA*n*）に伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_IICAn}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_IICAn}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_IICAn}_interrupt(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_IICA*n*}_master_handler

マスタ・モードでの IICA*n* データの送受信およびエラーを制御します。

備考　この API 関数は、IICA*n* 通信割り込みに伴う処理 r_{Config_ IICA*n*}_interrupt のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_IICAn}_master_handler(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_IICA*n*}_callback_master_sendend

マスタ送信完了の検出に伴う処理を実行します。

備考 1　この API 関数は、割り込み処理 r_{Config_IICA*n*}_master_handler のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアを維持し、他の処理コードをコールバックおよび割り込み関数から移動することに注意してください。それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_IICA*n*}_callback_master_sendend(void);

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_IICA*n*}_callback_master_receiveend

マスタ受信完了の検出に伴う処理を実行します。

備考 1　この API 関数は、割り込み処理 r_{Config_IICA*n*}_master_handler のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアを維持し、他の処理コードをコールバックおよび割り込み関数から移動することに注意してください。それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_IICA*n*}_callback_master_receiveend(void);

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_IICA*n*}_callback_master_error

マスタ・モードでのエラー検出に伴う処理を実行します。

備考 1　この API 関数は、割り込み処理 r_{Config_IICA*n*}_master_handler のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアを維持し、他の処理コードをコールバックおよび割り込み関数から移動することに注意してください。それ以外は、割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_IICA*n*}_callback_master_error(MD_STATUS flag);

備考　*n* はチャネル番号を示します。

[引数]

I/O	引数	説明
I	MD_STATUS flag;	ステータス・フラグ MD_NACK：NACK エラー

[戻り値]

なし

使用例

IICA1 マスタと EEPROM の通信例です（送信モードと受信モードの両方を含む）。

main.c (1/2)

```
#include "r_smc_entry.h"

#define EEPROM_ADDR   0x50    // 7 ビット EEPROM スレーブ・アドレス。
#define MEM_ADDR      0x00    // EEPROM メモリ・アドレス。
#define TIMEOUT       100     // 通信のタイムアウト。

/* Write buffer: memory address + 4 data bytes */
uint8_t write_data[5] = {MEM_ADDR, 0x11, 0x22, 0x33, 0x44};
uint8_t read_data[4] = {0}; // 4 データバイトを読み返す。

volatile uint8_t transmitend_flag = 0U;
volatile uint8_t receiveend_flag = 0U;

void main(void)
{
    EI(); // グローバル割り込みを有効にします。

    /* Check if bus is free */
    if (R_Config_IICA1_Bus_Check() != 0U)
    {
        R_Config_IICA1_Wait_Time(); // バスがビジーの場合は待ちます。
    }

    /* ---- Write to EEPROM ---- */
    R_Config_IICA1_StartCondition(); // Generate START condition
    if (R_Config_IICA1_Master_Send(EEPROM_ADDR, MEM_ADDR, write_data, sizeof(write_data),
        TIMEOUT) == 0U)
    {
        // 書き込み動作を正常に開始。
    }

    /* Wait for transmit complete using flag */
    while (transmitend_flag == 0U);
    transmitend_flag = 0U;

    /* Poll communication state until complete */
    while (R_Config_IICA1_Check_Comstate() != 0U)
    {
        R_Config_IICA1_Poll();
    }
    R_Config_IICA1_Wait_Comend(1U); // ストップ・コンディション。

    /* ---- Read from EEPROM ---- */
    R_Config_IICA1_StartCondition(); // Generate START condition again
    if (R_Config_IICA1_Master_Receive(EEPROM_ADDR, MEM_ADDR, read_data,
        sizeof(read_data), TIMEOUT) == 0U)
    {
        // 読み取り動作を正常に開始。
    }
```

main.c (2/2)

```
    /* Wait for receive complete using flag */
    while (receiveend_flag == 0U);
    receiveend_flag = 0U;

    /* Poll communication state until complete */
    while (R_Config_IICA1_Check_Comstate() != 0U)
    {
        R_Config_IICA1_Poll();
    }
        R_Config_IICA1_Wait_Comend(1U); // ストップ・コンディション。

    /* Stop IICA1 channel */
    R_Config_IICA1_Stop();

    /* Verify received data */
    if (read_data[0] == 0x11 && read_data[1] == 0x22 && read_data[2] == 0x33 && read_data[3] ==
0x44)
    {
        // データ検証に成功。
    }

    while (1)
    {
        // メイン・ループです。
    }
}
```

Config_IICA1_user.c

```
/* Start user code for global. Do not edit comment generated here */
extern volatile uint8_t transmitend_flag;
extern volatile uint8_t receiveend_flag;
/* End user code. Do not edit comment generated here */

static void r_Config_IICA1_callback_slave_sendend(void)
{
    /* Start user code for r_Config_IICA1_callback_slave_sendend. Do not edit comment generated
here */
    transmitend_flag = 1U;    // マスタ送信完了を示すフラグを設定します。
    /* End user code. Do not edit comment generated here */
}

static void r_Config_IICA1_callback_slave_receiveend(void)
{
    /* Start user code for r_Config_IICA1_callback_slave_receiveend. Do not edit comment
generated here */
    receiveend_flag = 1U;    // マスタ受信完了を示すフラグを設定します。
    /* End user code. Do not edit comment generated here */
}
```

4.2.57 IIC 通信 (スレーブ・モード) (シリアル・インタフェース IICA)

以下に、スマート・コンフィグレータが IIC 通信 (スレーブ・モード) (シリアル・インタフェース IICA) 用として出力する API 関数の一覧を示します。

表 4.63 IIC 通信 (スレーブ・モード) (シリアル・インタフェース IICA) 用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_IICA*n*}_Create	シリアル・インタフェース IICA	IICA*n* スレーブ・モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_IICA*n*}_Stop		IICAn スレーブの動作を終了します。
R_{Config_IICA*n*}_Slave_Send		スレーブ・モードでデータ送信を開始します。
R_{Config_IICA*n*}_Slave_Receive		スレーブ・モードでデータ受信を開始します。
R_{Config_IICAn}_Set_WakeupOn		STOP モード時のアドレス一致ウエイクアップ機能の動作を許可します。
R_{Config_IICAn}_Set_WakeupOff		STOP モード時のアドレス一致ウエイクアップ機能の動作を停止します。
R_{Config_IICA*n*}_Create_UserInit		IICAn に関するユーザ独自の初期化処理を実行します。
r_{Config_IICA*n*}_interrupt		IICA*n* 通信割り込み（INTIICA*n*）に伴う処理を実行します。
r_{Config_IICA*n*}_slave_handler		スレーブ・モードでの IICA*n* データの送受信およびエラーを制御します。
r_{Config_IICA*n*}_callback_slave_sendend		スレーブ送信完了の検出に伴う処理を実行します。
r_{Config_IICA*n*}_callback_slave_receiveend		スレーブ受信完了の検出に伴う処理を実行します。
r_{Config_IICA*n*}_callback_slave_error		スレーブ・モードでのエラー検出に伴う処理を実行します。
r_{Config_IICA*n*}_callback_getstopcondition		スレーブ停止条件の取得に伴う処理を実行します。

R_{Config_IICA*n*}_Create

IICA*n* スレーブ・モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_IICAn}_Create(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_IICA*n*}_Stop

IICA*n* モジュールの動作を終了します。

[指定形式]

```
void    R_{Config_IICAn}_Stop(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_IICA*n*}_Slave_Send

スレーブ・モードでデータ送信を開始します。

備考　通信完了時に停止条件を出さずにマスタを再起動した場合は、スレーブ・デバイスの対応するスレーブ機能の呼び出しに注意してください。
例えば、マスタ・デバイスでは、R_ {Config_IICAn} _Master_Receive 関数が呼び出されて通信が再開され、R_{Config_IICAn}_Slave_Send 関数がスレーブ・デバイスで呼び出されます。つまり、マスタ API とスレーブ API をペアで呼び出す必要があり、それ以外の場合、IICA 操作は保証されません。

[指定形式]

```
void    R_{Config_IICAn}_Slave_Send(uint8_t * const tx_buf, uint16_t tx_num);
```

備考　*n* はチャネル番号を示します。

[引数]

I/O	引数	説明
I	uint8_t * const tx_buf;	送信するデータを格納したバッファへのポインタ
I	uint16_t tx_num;	送信するバイト数

[戻り値]

なし

R_{Config_IICA*n*}_Slave_Receive

スレーブ・モードでデータ受信を開始します。

備考　通信完了時に停止条件を出さずにマスタを再起動した場合は、スレーブ・デバイスの対応するスレーブ機能の呼び出しに注意してください。
例えば、マスタ・デバイスでは、R_ {Config_IICAn} _Master_Receive 関数が呼び出されて通信が再開され、R_{Config_IICAn}_Slave_Send 関数がスレーブ・デバイスで呼び出されます。つまり、マスタ API とスレーブ API をペアで呼び出す必要があり、それ以外の場合、IICA 操作は保証されません。

[指定形式]

```
void    R_{Config_IICAn}_Slave_Receive(uint8_t * const rx_buf, uint16_t rx_num);
```

備考　*n* はチャネル番号を示します。

[引数]

I/O	引数	説明
O	uint8_t * const rx_buf;	受信したデータを格納したバッファへのポインタ
O	uint16_t rx_num;	受信するバイト数

[戻り値]

なし

R_{Config_IICA*n*}_Set_WakeupOn

STOP モード時のアドレス一致ウエイクアップ機能の動作を許可します。

[指定形式]

```
void    R_{Config_IICAn}_Set_WakeupOn(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_IICA*n*}_Set_WakeupOff

STOP モード時のアドレス一致ウエイクアップ機能の動作を停止します。

[指定形式]

```
void    R_{Config_IICAn}_Set_WakeupOff(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_IICA*n*}_Create_UserInit

IICA*n* に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_IICA*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_IICAn}_Create_UserInit(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_IICA*n*}_interrupt

IICA*n* 通信割り込み（INTIICA*n*）に伴う処理を実行します。

［指定形式］

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_IICAn}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_IICAn}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_IICAn}_interrupt(void);
```

備考　　*n* はチャネル番号を示します。

［引数］

なし

［戻り値］

なし

r_{Config_IICA*n*}_slave_handler

スレーブ・モードでの IICA*n* データの送受信およびエラーを制御します。

備考 1　この API 関数は、IICA*n* 通信割り込みに伴う処理 r_{Config_ IICA*n*}_interrupt のコールバック・ルーチンとして呼び出されます。

備考 2　スマート・コンフィグレータは、g_iican_slave_status_flag を使用してユーザープログラムフローを制御します。g_iica0_slave_status_flag の初期化は、R_{Config_IICAn}_Slave_Send 関数と R_{Config_IICAn}_Slave_Receive 関数にあります。

[指定形式]

```
void    R_{Config_IICAn}_slave_handler(void);
```

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_IICA*n*}_callback_slave_sendend

スレーブ送信完了の検出に伴う処理を実行します。

備考 1　この API 関数は、割り込み処理 r_{Config_IICAn}_slave_handler のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアを維持し、他の処理コードをコールバックおよび割り込み関数から移動することに注意してください。 それ以外は、次の割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_IICA*n*}_callback_slave_sendend(void);

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_IICA*n*}_callback_slave_receiveend

スレーブ受信完了の検出に伴う処理を実行します。

備考 1　この API 関数は、割り込み処理 r_{Config_IICAn}_slave_handler のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアを維持し、他の処理コードをコールバックおよび割り込み関数から移動することに注意してください。 それ以外は、次の割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_IICA*n*}_callback_slave_receiveend(void);

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

r_{Config_IICA*n*}_callback_slave_error

スレーブ・モードでのエラー検出に伴う処理を実行します。

備考 1　この API 関数は、割り込み処理 r_{Config_IICAn}_slave_handler のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアを維持し、他の処理コードをコールバックおよび割り込み関数から移動することに注意してください。 それ以外は、次の割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_IICA*n*}_callback_slave_error(MD_STATUS flag);

備考　*n* はチャネル番号を示します。

[引数]

I/O	引数	説明
I	MD_STATUS flag;	ステータス・フラグ MD_NACK：ACK エラー検出

[戻り値]

なし

r_{Config_IICA*n*}_callback_getstopcondition

スレーブ停止条件の取得に伴う処理を実行します。

備考 1　この API 関数は、割り込み処理 r_{Config_IICAn}_slave_handler のコールバック・ルーチンとして呼び出されます。

備考 2　コールバック関数で必要なフラグの設定／クリアを維持し、他の処理コードをコールバックおよび割り込み関数から移動することに注意してください。 それ以外は、次の割り込みが正しいタイミングで処理されません。

[指定形式]

static void	r_{Config_IICA*n*}_callback_getstopcondition(void);

備考　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

使用例

シリアル・アレイ・ユニット IIC マスタ・モードの使用例を参照してください。

4.2.58 割り込みコントローラ

以下に、スマート・コンフィグレータが割り込みコントローラ用として出力する API 関数の一覧を示します。

表 4.64 割り込みコントローラ用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_INTC}_Create	割り込み機能	INTC モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_INTC}_INTP*n*_Start		INTP*n* 割り込みフラグをクリアし、割り込みを許可します。
R_{Config_INTC}_INTP*n*_Stop		INTP*n* 割り込みを禁止し、割り込みフラグをクリアします。
R_{Config_INTC}_Create_UserInit		INTC モジュールに関するユーザ独自の初期化処理を実行します。
r_{Config_INTC}_intpn_interrupt		INTP*n* 割り込みに伴う処理を実行します。

R_{Config_INTC}_Create

INTC モジュールを制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_INTC}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_INTC}_INTP*n*_Start

INTP*n* 割り込みフラグをクリアし、割り込みを許可します。

[指定形式]

```
void    R_{Config_INTC}_INTPn_Start(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_INTC}_INTP*n*_Stop

INTP*n* 割り込みを禁止し、割り込みフラグをクリアします。

[指定形式]

```
void    R_{Config_INTC}_INTPn_Stop(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_INTC}_Create_UserInit

INTC コントローラに関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_INTC}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_INTC}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_INTC}_intp*n*_interrupt

INTP*n* 割り込みに伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_INTC}_intpn_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_INTC}_intpn_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_INTC}_intpn_interrupt(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

使用例

INTP0 の有効なエッジ入力を検出するときにフラグを設定する例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t intp0_int_flag;
void main(void);

void main(void)
{
	EI();	// グローバル割り込みを有効にして割り込み処理を許可します。
	intp0_int_flag = 0U;	// INTP0 割り込みフラグを 0 に初期化します。
	R_Config_INTC_INTP0_Start ();	// INTP0 割り込み検出を開始します。
	while(intp0_int_flag != 1U);	// NTP0 割り込みフラグが設定されるまで待ちます。
	R_Config_ INTC_INTP0_Stop ();	// INTP0 割り込み検出を停止します。
}
```

Config_INTC_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t intp0_int_flag = 0U;
/* End user code. Do not edit comment generated here */

static void __near r_Config_INTC_intp0_interrupt(void)
{
	/* Start user code for r_Config_INTC_intp0_interrupt. Do not edit comment generated here */
	/* Set the flag */
	intp0_int_flag = 1U;	// 有効エッジの検出時に INTP0 割り込みフラグを設定します。
	/* End user code. Do not edit comment generated here */
}
```

4.2.59 電圧検出回路

以下に、スマート・コンフィグレータが電圧検出回路用として出力する API 関数の一覧を示します。

表 4.65 電圧検出回路用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_LVD*n*}_Create	電圧検出回路	電圧検出回路モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_LVD*n*}_Start		電圧検出回路の動作を開始します。
R_{Config_LVD*n*}_Stop		電圧検出回路の動作を終了します。
R_{Config_LVD*n*}_Create_UserInit		電圧検出回路に関するユーザ独自の初期化処理を実行します。

R_{Config_LVD*n*}_Create

電圧検出回路モジュールを制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_LVDn}_Create(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_LVD*n*}_Start

電圧検出回路の動作を開始します。

[指定形式]

void R_{Config_LVD*n*}_Start(void);

備考 *n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_LVD*n*}_Stop

電圧検出回路の動作を終了します。

[指定形式]

```
void    R_{Config_LVDn}_Stop(void);
```

備考　　*n* はチャネル番号を示します。

[引数]

なし

[戻り値]

なし

R_{Config_LVD*n*}_Create_UserInit

電圧検出回路に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_LVD*n*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_LVDn}_Create_UserInit(void);
```

備考　*n* はユニット番号を示します。

[引数]

なし

[戻り値]

なし

使用例

割り込みモードで動作する LVD の例です。

main.c

```
#include "r_smc_entry.h"

void main(void);

void main(void)
{
    EI();    // グローバル割り込みを有効にして割り込み処理を許可します。
    R_LVD_Start_Interrupt();    // LVD を割り込みモードで起動します。
    R_ Config_LVD1_Start();    // LVD1 モジュールの設定を開始します。
}
```

r_cg_lvd_common_user.c

```
static void __near r_lvd_interrupt(void)
{
    /* Start user code for r_lvd_interrupt. Do not edit comment generated here */
    /*Clear Flag*/
    DLVD1FCLR = 1U;    // フラグをクリアするには、DLVD1FCLR ビットに 1 を書き込みます。
    /* End user code. Do not edit comment generated here */
}
```

4.2.60 SNOOZE モード・シーケンサ

以下に、スマート・コンフィグレータが SNOOZE モード・シーケンサ用として出力する API 関数の一覧を示します。

表 4.66 SNOOZE モード・シーケンサ用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_SMS}_Create	SNOOZE モード・シーケンサ	SMS の構成、SMS 命令のコピー、SMS データのコピーなど、SMS モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_SMS}_Start		引数から SMS データを設定し、SMS モジュールの動作を開始します。
R_{Config_SMS}_Stop		SMS モジュールの動作を終了します。
R_{Config_SMS}_GetStatus		SMS ウエイクアップ・ステータスをチェックします。
R_{Config_SMS}_GetReturn		SMS データを返します。
R_{Config_SMS}_TriggerWait_Disable		トリガ・ウエイト動作を禁止します。
R_{Config_SMS}_TriggerWait_Enable		トリガ・ウエイト動作を許可します。
R_{Config_SMS}_Set_PowerOn		SMS モジュールへのクロック供給を開始します。
R_{Config_SMS}_Set_PowerOff		SMS モジュールへのクロック供給を停止します。
R_{Config_SMS}_Set_Reset		SMS モジュールをリセット状態にします。
R_{Config_SMS}_Release_Reset		SMS モジュールのリセット状態を解除します。
R_{Config_SMS}_Create_UserInit		SMS モジュールに関するユーザ独自の初期化処理を実行します。
r_{Config_SMS}_interrupt		INTSMSE 割り込みに伴う処理を実行します。

R_{Config_SMS}_Create

SMS の構成、SMS 命令のコピー、SMS データのコピーなど、SMS モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_SMS}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_SMS}_Start

引数から SMS データを設定し、SMS モジュールの動作を開始します。

[指定形式]

```
void    R_{Config_SMS}_Start(void);
```

```
void    R_{Config_SMS}_Start(uint16_t arg1, uint16_t arg2, ……, uint16_t argn);
```

備考 1. 本 API 関数の引数の数は、Start Block の設定により異なります。たとえば、Start Block 設定に 3 つの引数があれば、この API 関数は R_{Config_SMS}_Start(uint16_t arg1, uint16_t arg2, uint16_t arg3)となります。

備考 2. *n*≦14

[引数]

I/O	引数	説明
I	uint16_t arg*n*;	SMS 開始データ （n≦14）

備考 *n*＝1～14

[戻り値]

なし

R_{Config_SMS}_Stop

SMS モジュールの動作を終了します。

[指定形式]

```
void    R_{Config_SMS}_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_SMS}_GetStatus

SMS ウエイクアップ・ステータスをチェックします。

［指定形式］

```
uint8_t    R_{Config_SMS}_GetStatus(void);
```

［引数］

なし

［戻り値］

マクロ	説明
g_sms_finish_flag	SMS 終了フラグ 0：ウエイクアップしない 1：ウエイクアップする

R_{Config_SMS}_GetReturn

SMS データを返します。

[指定形式]

void R_{Config_SMS}_GetReturn(void);

void R_{Config_SMS}_GetReturn(uint16_t *p_ret1, uint16_t *p_ret2, ….., uint16_t *p_ret*n*);

備考 1. 本 API 関数の引数の数は、Wake Up Block の設定により異なります。たとえば、Wake Up Block 設定に 2 つの引数があれば、この API 関数は R_{Config_SMS}_GetReturn(uint16_t *p_ret1, uint16_t *p_ret2)となります。

備考 2. *n*≦14

[引数]

I/O	引数	説明
I	uint16_t *p_ret*n*;	SMS が返すデータ（n≦14）

備考 *n*＝1～14

[戻り値]

なし

R_{Config_SMS}_TriggerWait_Disable

トリガ・ウエイト動作を禁止します。

[指定形式]

```
void    R_{Config_SMS}_TriggerWait_Disable(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_SMS}_TriggerWait_Enable

トリガ・ウエイト動作を許可します。

[指定形式]

```
void    R_{Config_SMS}_TriggerWait_Enable(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_SMS}_Set_PowerOn

SMS モジュールへのクロック供給を開始します。

[指定形式]

```
void    R_{Config_SMS}_Set_PowerOn(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_SMS}_Set_PowerOff

SMS モジュールへのクロック供給を停止します。

[指定形式]

```
void    R_{Config_SMS}_Set_PowerOff(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_SMS}_Set_Reset

SMS モジュールをリセット状態にします。

[指定形式]

```
void    R_{Config_SMS}_Set_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_SMS}_Release_Reset

SMS モジュールのリセット状態を解除します。

[指定形式]

```
void    R_{Config_SMS}_Release_Reset(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_SMS}_Create_UserInit

SMS モジュールに関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_SMS}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_SMS}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_SMS}_interrupt

INTSMSE 割り込みに伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_SMS}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_SMS}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_SMS}_interrupt(void);
```

[引数]

なし

[戻り値]

なし

使用例

SMS を使用して A/D 変換結果を読み取る例です。

main.c

```
#include "r_smc_entry.h"

uint8_t g_sms_finish_flag;    // SMS 動作の完了を示すフラグです。
uint16_t adc_single_chl = 0x7890;    // A/D 変換結果を格納する変数です。

void main(void);

void main(void)
{
      EI();    // グローバル割り込みを有効にして割り込み処理を許可します。

      // SMS をシーケンサ停止状態に設定します。
      R_Config_ SMS_TriggerWait_Enable();
      R_Config_SMS_Start ((uint16_t)&adc_single_chl, (uint16_t)&adc_single_chl);     // 送信元と送信先
の両方として adc_single_chl のアドレスを使用して SMS 動作を開始します。
      // SMS をシーケンサ動作状態に設定します。
      R_Config_ SMS_TriggerWait_Disable();
      STOP ();    // 割り込みが発生するまで低電力モードに入ります。
      while (g_sms_finish_flag != 1);  // SMS 動作が完了するまで待ちます。
      g_sms_finish_flag = 0;    // 次の動作のために完了フラグをリセットします。
      R_Config_SMS_Stop ();    // SMS 動作を停止します。
}
```

Config_SMS_user.c

```
/* Start user code for global. Do not edit comment generated here */
extern volatile uint8_ t  g_sms_finish_flag;
/* End user code. Do not edit comment generated here */

static void __near r_Config_SMS_interrupt(void)
{
      /* Start user code for r_Config_SMS_interrupt. Do not edit comment generated here */
      g_sms_finish_flag = 1;    // SMS 割り込み発生時に完了フラグを設定します。
      /* End user code. Do not edit comment generated here */
}
```

4.2.61 キー割り込み

以下に、スマート・コンフィグレータがキー割り込み用として出力する API 関数の一覧を示します。

表 4.67 キー割り込み用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_KR}_Create	キー割り込み	キー割り込みモジュールを制御する前に必要な初期化処理を実行します。
R_{Config_KR}_Start		INTKR 割り込みフラグをクリアし、割り込みを許可します。
R_{Config_KR}_Stop		INTKR 割り込みを禁止し、割り込みフラグをクリアします。
R_{Config_KR}_Create_UserInit		キー割り込み機能に関するユーザ独自の初期化処理を実行します。
r_{Config_KR}_interrupt		INTKR 割り込みに伴う処理を実行します。

R_{Config_KR}_Create

キー割り込みモジュールを制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_KR}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_KR}_Start

INTKR 割り込みフラグをクリアし、割り込みを許可します。

[指定形式]

```
void    R_{Config_KR}_Start(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_KR}_Stop

INTKR 割り込みを禁止し、割り込みフラグをクリアします。

[指定形式]

```
void    R_{Config_KR}_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_KR}_Create_UserInit

キー割り込み機能に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_KR}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_KR}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_KR}_interrupt

INTKR 割り込みに伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_KR}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_KR}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_KR}_interrupt(void);
```

[引数]

なし

[戻り値]

なし

使用例

KR 割り込みを検出するときにフラグを設定するための例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t kr_int_flag;
void main(void);

void main(void)
{
    EI();    // グローバル割り込みを有効にして割り込み処理を許可します。
    kr_int_flag = 0U;    // KR 割り込みフラグを 0 に初期化します。
    R_Config_KR_Start ();    // KR 割り込み検出モジュールを起動します。
    while(kr_int_flag != 1U);    // KR 割り込みフラグが 1 に設定されるまでループで待機します。
    R_Config_ KR_Stop ();    // KR 割り込み検出モジュールを停止します。
}
```

Config_KR_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t kr_int_flag = 0U;
/* End user code. Do not edit comment generated here */

static void __near r_Config_KR_interrupt(void)
{
    /* Start user code for r_Config_KR_interrupt. Do not edit comment generated here */
    /* Set the flag */
    kr_int_flag = 1U;    // 割り込み発生時に KR 割り込みフラグを 1 に設定します。
    /* End user code. Do not edit comment generated here */
}
```

4.2.62 リモコン信号受信機能

以下に、スマート・コンフィグレータがリモコン信号受信機能用として出力する API 関数の一覧を示します。

表 4.68 リモコン信号受信機能用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_REMC}_Create	リモコン信号受信機能	REMC ジュールを制御する前に必要な初期化処理を実行します。
R_{Config_REMC}_Start		リモコン信号受信機能の動作を開始します。
R_{Config_REMC}_Stop		リモコン信号受信機能の動作を停止します。
R_{Config_SMS}_Read		受信データの格納先と受信するバイト数を指定します。
R_{Config_REMC}_Create_UserInit		リモコン信号受信機能に関するユーザ独自の初期化処理を実行します。
r_{Config_REMC}_interrupt		INTREMC 割り込みに伴う処理を実行します。
r_{Config_REMC}_callback_receiveend		データ受信完了割り込みに伴う処理を実行します。
r_{Config_REMC}_callback_comparematch		コンペアー致割り込みに伴う処理を実行します。
r_{Config_REMC}_callback_receiveerror		受信エラー割り込みに伴う処理を実行します。
r_{Config_REMC}_callback_bufferfull		受信バッファ・フル割り込みに伴う処理を実行します。
r_{Config_REMC}_callback_header		ヘッダ・パターン一致割り込みに伴う処理を実行します。
r_{Config_REMC}_callback_data0		データ"0"パターン一致割り込みに伴う処理を実行します。
r_{Config_REMC}_callback_data1		データ"1"パターン一致割り込みに伴う処理を実行します。
r_{Config_REMC}_callback_specialdata		特殊データ・パターン一致割り込みに伴う処理を実行します。

R_{Config_REMC}_Create

REMC ジュールを制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_REMC}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_REMC}_Start

リモコン信号受信機能の動作を開始します。

[指定形式]

```
void    R_{Config_REMC}_Start(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_REMC}_Stop

リモコン信号受信機能の動作を停止します。

[指定形式]

```
void    R_{Config_REMC}_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_SMS}_Read

受信データの格納先と受信するバイト数を指定します。

備考　この API 関数は、データ受信の最後に REMC 割り込みによって読み取られた受信データの保存先を指定します。

[指定形式]

MD_STATUS　R_{Config_SMS}_Read(uint8_t * const rx_buf, uint8_t rx_num);

[引数]

I/O	引数	説明
O	uint8_t * const rx_buf;	受信したデータを格納したバッファへのポインタ
O	Uint8_t rx_num;	受信するバイト数

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_ERROR1	引数 rx_num の指定が無効

R_{Config_REMC}_Create_UserInit

リモコン信号受信機能に関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_REMC}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_REMC}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_REMC}_interrupt

INTREMC 割り込みに伴う処理を実行します。

［指定形式］

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_REMC}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_REMC}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_REMC}_interrupt(void);
```

［引数］

なし

［戻り値］

なし

r_{Config_REMC}_callback_receiveend

データ受信完了割り込みに伴う処理を実行します。

備考　この API 関数は、INTREMC 割り込みに伴う処理 r_{Config_REMC}_interrupt のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
static void    r_{Config_REMC}_callback_receiveend(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_REMC}_callback_comparematch

コンペアー致割り込みに伴う処理を実行します。

備考　この API 関数は、INTREMC 割り込みに伴う処理 r_{Config_REMC}_interrupt のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
static void    r_{Config_REMC}_callback_comparematch(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_REMC}_callback_receiveerror

受信エラー割り込みに伴う処理を実行します。

備考　この API 関数は、INTREMC 割り込みに伴う処理 r_{Config_REMC}_interrupt のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
static void    r_{Config_REMC}_callback_receiveerror(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_REMC}_callback_bufferfull

受信バッファ・フル割り込みに伴う処理を実行します。

備考　この API 関数は、INTREMC 割り込みに伴う処理 r_{Config_REMC}_interrupt のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
static void    r_{Config_REMC}_callback_bufferfull(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_REMC}_callback_header

ヘッダ・パターン一致割り込みに伴う処理を実行します。

備考　この API 関数は、INTREMC 割り込みに伴う処理 r_{Config_REMC}_interrupt のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
static void    r_{Config_REMC}_callback_header(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_REMC}_callback_data0

データ"0"パターン一致割り込みに伴う処理を実行します。

備考　この API 関数は、REMC 割り込みに伴う処理 r_{Config_REMC}_interrupt のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
static void    r_{Config_REMC}_callback_data0(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_REMC}_callback_data1

データ"1"パターン一致割り込みに伴う処理を実行します。

備考　この API 関数は、INTREMC 割り込みに伴う処理 r_{Config_REMC}_interrupt のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
static void    r_{Config_REMC}_callback_data1(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_REMC}_callback_specialdata

特殊データ・パターン一致割り込みに伴う処理を実行します。

備考　この API 関数は、INTREMC 割り込みに伴う処理 r_{Config_REMC}_interrupt のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
static void    r_{Config_REMC}_callback_specialdata(void);
```

[引数]

なし

[戻り値]

なし

使用例

データ受信終了時にリモコン信号受信機能の動作を停止する例です。

main.c

```
#include "r_smc_entry.h"

volatile uint8_t g_remc_rx_buf[8];    // 受信したリモコン信号データを格納する揮発性バッファを宣言します

void main(void);

void main(void)
{
    EI();    // グローバル割り込みを有効にして割り込み処理を許可します。
    /* Start the REMC operation */
    R_Config_REMC_Start();

    /* Read data from receive data buffer */
    R_Config_REMC_Read((uint8_t  *)g_remc_rx_buf,  8U);       // 受信したデータの 8 バイトを g_remc_rx_buf に読み込むます。
}
```

Config_REMC_user.c

```
static void r_Config_REMC_callback_receiveend(void)
{
    /* Start user code for r_Config_REMC_callback_receiveend. Do not edit comment generated here */
    R_Config_REMC_Stop();    // データ受信完了後に REMC 動作を停止します。
    /* End user code. Do not edit comment generated here */
}
```

4.2.63 ウォッチドッグ・タイマ

以下に、スマート・コンフィグレータがウォッチドッグ・タイマ用として出力する API 関数の一覧を示します。

表 4.69 ウォッチドッグ・タイマ用 API 関数

API 関数名	周辺機能	機能概要
R_{Config_WDT}_Create	ウォッチドッグ・タイマ	ウォッチドッグ・タイマ・モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_WDT}_Restart		ウォッチドッグ・タイマのカウンタをクリアしてから、カウントを再開します。
R_{Config_WDT}_Create_UserInit		ウォッチドッグ・タイマに関するユーザ独自の初期化処理を実行します。
r_{Config_WDT}_interrupt		INTWDTI 割り込みに伴う処理を実行します。

R_{Config_WDT}_Create

ウォッチドッグ・タイマ・モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、main()関数を実行する前に、R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_WDT}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_WDT}_Restart

ウォッチドッグ・タイマのカウンタをクリアしてから、カウントを再開します。

[指定形式]

```
void    R_{Config_WDT}_Restart(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_WDT}_Create_UserInit

ウォッチドッグ・タイマに関するユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_WDT}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_WDT}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_WDT}_interrupt

INTWDTI 割り込みに伴う処理を実行します。

備考　この API 関数は、オーバーフロー時間の 75% + 1/4 fIL に達したときに、マスカブル割り込みの割り込みハンドラとして呼び出されます。

［指定形式］

CCRL78 ツールチェーンの場合

```
static void __near    r_{Config_WDT}_interrupt(void);
```

LLVM ツールチェーンの場合

```
void    r_{Config_WDT}_interrupt(void);
```

IAR ツールチェーンの場合

```
__interrupt static void    r_{Config_WDT}_interrupt(void);
```

［引数］

なし

［戻り値］

なし

使用例

メイン関数のすべてのループでカウンタ値を更新し、カウンタのアンダフローに対してソフトウェア・リセットを発行する例です。

main.c

```
#include "r_smc_entry.h"

void main(void);

void main(void)
{
    EI();    // グローバル割り込みを有効にして割り込み処理を許可します。
    while (1U)    // ウォッチドッグ・タイマを継続的に更新するループです。
    {
        /* Restarts WDT module */
        R_Config_WDT_Restart();    // カウンタのアンダフローに応答するために、ソフトウェア・リセットを介してウォッチドッグ・タイマを更新します。
    }
}
```

4.2.64 ロジック&イベント・リンク・コントローラ (ELCL)

ロジック&イベント・リンク・コントローラ (ELCL) コンポーネントには 2 種類あり、1 種類は「ELCL チャタリング防止機能」「ELCL スレーブセレクト端子機能」などの固定機能 ELCL コンポーネントで、もう 1 種類は「ELCL Flexible Circuit」です。2 種類の ELCL コンポーネント の API は同じではありません。

以下に、スマート・コンフィグレータがロジック&イベント・リンク・コントローラ (ELCL) 用として出力する API 関数の一覧を示します。

表 4.70 ロジック&イベント・リンク・コントローラ (ELCL) API 関数

API 関数名	周辺機能	機能概要
R_{Config_xxx}_Create	ロジック&イベント・リンク・コントローラ	ELCL モジュールを制御する前に必要な初期化処理を実行します。
R_{ConfigL_xxx}_Start		ELCL 出力を有効にします。
R_{Config_xxx}_Stop		ELCL 出力を無効にします。
R_{Config_xxx}_OUTPUTn_Start		シグナル [*n*] の ELCL 出力を開始します。
R_{Config_xxx}_OUTPUTn_Stop		シグナル [*n*] の ELCL 出力を停止します。
R_{Config_xxx}_GetStatus		ELOENCTL レジスタ値を取得して、どの出力が有効になっているかを確認します。
R_{Config_xxx}_Create_UserInit		ELCL のユーザ独自の初期化処理を実行します。
r_{Config_xxx}_interrupt		INTELCL 割り込みに応答して処理を実行します。

注1 *xxx* は ELCL モジュール名です。

注2 R_ {Config_xxx} _interrupt 関数は、ELCL コンポーネントで ELCL 出力信号が INTELCL として使用されている場合にのみ生成されます。

注3 ELCL Flexible Circuit コンポーネントのみが、R_{Config_xxx}_OUTPUTn_Start、R_{Config_xxx}_OUTPUTn_Stop R_{Config_xxx}_GetStatus を生成します。固定機能の ELCL コンポーネントでは、これらの API 関数を生成しません。

R_{Config_*xxx*}_Create

ELCL モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、main() 関数が実行される前に R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_xxx}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_*xxx*}_Start

ELCL 割り込みフラグをクリアし、ELCL 出力を有効にします。

[指定形式]

```
void    R_{Config_xxx}_Start(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_*xxx*}_Stop

INTELCL 割り込みの無効および割り込みフラグをクリアし、ELCL 出力を無効にします。

[指定形式]

```
void    R_{Config_xxx}_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_*xxx*}_OUTPUT*n*_Start

シグナル [*n*] の ELCL 出力を開始します。

[指定形式]

```
void    R_{Config_xxx}_OUTPUT_Stat(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_*xxx*}_OUTPUT*n*_Stop

シグナル [*n*] の ELCL 出力を停止します。

[指定形式]

```
void    R_{Config_xxx}_OUTPUTn_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_*xxx*}_GetStatus

ELOENCTL レジスタ値を取得して、どの出力が有効になっているかを確認します。

[指定形式]

```
uint8_t    R_{Config_xxx}_GetStatus(void);
```

[引数]

I/O	引数	説明
I	Uint8_t status;	ELOENCTL レジスタ値

[戻り値]

なし

R_{Config_*xxx*}_Create_UserInit

ELCL のユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_*xxx*}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_xxx}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_*xxx*}_interrupt

INTELCL 割り込みに応答して処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合:

```
static void __near    r_{ Config_xxx}_interrupt(void);
```

LLVM ツールチェーンの場合:

```
void    r_{Config_xxx}_interrupt(void);
```

IAR ツールチェーンの場合:

```
__interrupt static void    r_{ Config_xxx}_interrupt(void);
```

[引数]

なし

[戻り値]

なし

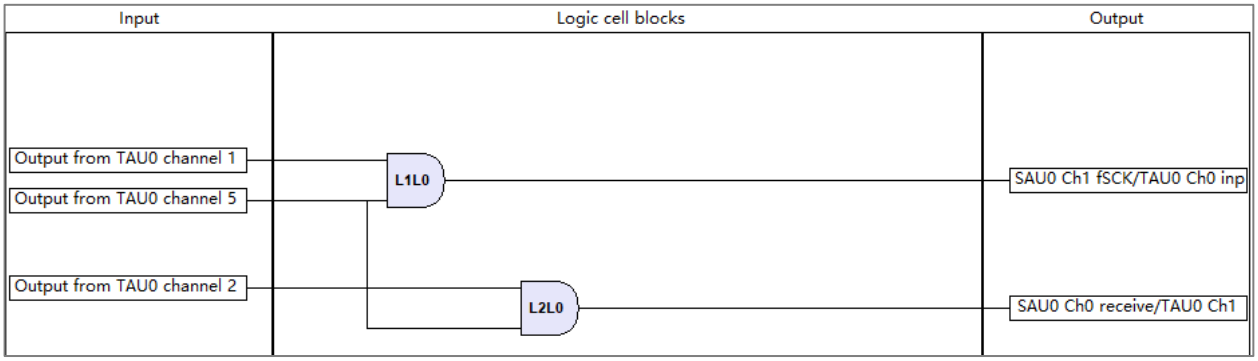
使用例

例 1：ELCL AND コンポーネントを使用して 2 ポートの単一 AND ロジックを実装する例です。

main.c

```
#include "r_smc_entry.h"

void main(void);

void main(void)
{
    EI();    // グローバル割り込みを有効にして割り込み処理を許可します。

    R_Config_AND_Start();    // 構成された AND ロジック・コンポーネントを起動します。

}
```

例 2：ELCL Flexible Circuit コンポーネントを使用して、以下の ELCL 機能を実装する例です。



main.c

```
#include "r_smc_entry.h"

void main(void);

void main(void)
{
    uint8_t status;
    EI();    // グローバル割り込みを有効にして割り込み処理を許可します。

    R_Config_ELCL_OUTPUT3_Start();    // ELCL 出力チャネル 3 を起動します。
    R_Config_ELCL_OUTPUT4_Start();    // ELCL 出力チャネル 4 を起動します。
  …

    status = R_Config_ELCL_GetStatus();    // ELCL 回路のステータスを取得します。
    while (1)
    {
        ;
    }
}
```

4.2.65 イベント・リンク・コントローラ

以下に、スマート・コンフィグレータがイベント・リンク・コントローラ用として出力する API 関数の一覧を示します。

表 4.71 イベント・リンク・コントローラ API 関数

API 関数名	周辺機能	機能概要
R_{Config_ELC}_Create	イベント・リンク・コントローラ	ELC モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_ELC}_Stop		ELC 出力を無効にします。
R_{Config_ELC}_Create_UserInit		ELC のユーザ独自の初期化処理を実行します。

R_{Config_ELC}_Create

ELC モジュールを制御する前に必要な初期化処理を実行します。

備考　この API 関数は、main() 関数が実行される前に R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_ELC}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_ELC}_Stop

INTELCL 割り込みの無効および割り込みフラグをクリアし、ELC 出力を無効にします。

[指定形式]

```
void    R_{Config_ELC}_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_ELC}_Create_UserInit

ELC のユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_ELC}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_ELC}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

使用例

ELC を使用して、キー・リターン信号検出イベントの生成を停止する例です。

main.c

```
#include "r_smc_entry.h"

void main(void);

void main(void)
{
    uint32_t stp_event = 0x00000020;    // 停止するイベント ID を表す変数を定義します。（キー・リターン信号検出）
    EI();   // グローバル割り込みを有効にして割り込み処理を許可します。
    R_Config_ELC_Stop(stp_even);    // 指定 ELC イベントを停止します。（キー・リターン信号検出）

    while(1);   // プログラムを実行し続けるためのループ。
    {
        ;
    }
}
```

4.2.66 LCD コントローラ／ドライバ

以下に、スマート・コンフィグレータが LCD コントローラ／ドライバ用として出力する API 関数の一覧を示します。

表 4.72 LCD コントローラ／ドライバ API 関数

API 関数名	周辺機能	機能概要
R_{Config_LCD}_Create	LCD コントローラ／ドライバ	LCD コントローラ／ドライバ・モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_LCD}_Start		LCD コントローラ／ドライバの動作を開始します。
R_{Config_LCD}_Stop		LCD コントローラ／ドライバの動作を停止します。
R_{Config_LCD}_Voltage_On		昇圧回路または容量分割回路の動作を許可します。
R_{Config_LCD}_Voltage_Off		昇圧回路または容量分割回路の動作を停止します。
R_{Config_LCD}_Set_DisplayData		LCD コントローラ／ドライバに表示するデータを設定します。
R_{Config_ELC}_Create_UserInit		LCD コントローラ／ドライバのユーザ独自の初期化処理を実行します。

R_{Config_LCD}_Create

LCD コントローラ／ドライバ・モジュールを制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、main() 関数が実行される前に R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_LCD}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_LCD}_Start

LCD コントローラ／ドライバの動作を開始します。

[指定形式]

```
void    R_{Config_LCD}_Start(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_LCD}_Stop

LCD コントローラ／ドライバの動作を停止します。

[指定形式]

```
void    R_{Config_LCD}_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_LCD}_Voltage_On

昇圧回路または容量分割回路の動作を許可します。

[指定形式]

```
void    R_{Config_LCD}_Voltage_On(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_LCD}_Voltage_Off

昇圧回路または容量分割回路の動作を停止します。

[指定形式]

```
void    R_{Config_LCD}_Voltage_Off(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_LCD}_Set_DisplayData

LCD コントローラ／ドライバに表示するデータを設定します。

[指定形式]

```
MD_STATUS    R_{Config_LCD}_Set_DisplayData(uint8_t index, uint8_t data);
```

[引数]

I/O	引数	説明
I	uint8_t index;	LCD 表示データ・レジスタ(SEG*n*)のインデックスを指定 (n=0～55)
I	uint8_t data;	表示するデータ

[戻り値]

マクロ	説明
MD_OK	正常終了
MD_ARGERROR	引数入力エラー

R_{Config_LCD}_Create_UserInit

LCD コントローラ／ドライバのユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_LCD}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_LCD}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

使用例

LCD コントローラ／ドライバの例です。

main.c (1/2)

```
#include "r_smc_entry.h"

void main(void);

void main(void)
{
    EI();   // グローバル割り込みを有効にして割り込み処理を許可します。
    // 桁 1 のセグメントデータを設定すると、桁 1 に数字 1 が表示されます。
    R_Config_LCD_Set_DisplayData(7, 0x00);
    R_Config_LCD_Set_DisplayData(8, 0x00);
    R_Config_LCD_Set_DisplayData(12, 0x00);
    R_Config_LCD_Set_DisplayData(13, 0x06);

    // 桁 2 のセグメントデータを設定すると、桁 2 に数字 2 が表示されます。
    R_Config_LCD_Set_DisplayData(6, 0x03);
    R_Config_LCD_Set_DisplayData(5, 0x02);
    R_Config_LCD_Set_DisplayData(14, 0x04);
    R_Config_LCD_Set_DisplayData(15, 0x0C);

    // 桁 3 のセグメントデータを設定すると、桁 3 に数字 3 が表示されます。
    R_Config_LCD_Set_DisplayData(4, 0x01);
    R_Config_LCD_Set_DisplayData(3, 0x02);
    R_Config_LCD_Set_DisplayData(16, 0x04);
    R_Config_LCD_Set_DisplayData(17, 0x0E);

    // 桁 4 のセグメントデータを設定すると、桁 4 に数字 4 が表示されます。
    R_Config_LCD_Set_DisplayData(2, 0x04);
    R_Config_LCD_Set_DisplayData(1, 0x02);
    R_Config_LCD_Set_DisplayData(20, 0x04);
    R_Config_LCD_Set_DisplayData(21, 0x06);

    // 桁 5 のセグメントデータを設定すると、桁 5 に数字 5 が表示されます。
    R_Config_LCD_Set_DisplayData(0, 0x05);
    R_Config_LCD_Set_DisplayData(50, 0x02);
    R_Config_LCD_Set_DisplayData(28, 0x04);
    R_Config_LCD_Set_DisplayData(51, 0x0A);

    // 桁 6 のセグメントデータを設定すると、桁 6 に数字 6 が表示されます。
    R_Config_LCD_Set_DisplayData(49, 0x07);
    R_Config_LCD_Set_DisplayData(48, 0x02);
    R_Config_LCD_Set_DisplayData(52, 0x04);
    R_Config_LCD_Set_DisplayData(53, 0x0A);

    // 桁 7 のセグメントデータを設定すると、桁 7 に数字 7 が表示されます。
    R_Config_LCD_Set_DisplayData(47, 0x00);
    R_Config_LCD_Set_DisplayData(39, 0x00);
    R_Config_LCD_Set_DisplayData(54, 0x00);
    R_Config_LCD_Set_DisplayData(55, 0x0E);
```

main.c (2/2)

```
        // 桁 8 のセグメントデータを設定すると、桁 8 に数字 8 が表示されます。
        R_Config_LCD_Set_DisplayData(37, 0x02);
        R_Config_LCD_Set_DisplayData(38, 0x07);
        R_Config_LCD_Set_DisplayData(35, 0x04);
        R_Config_LCD_Set_DisplayData(36, 0x0E);

        R_Config_LCD_Voltage_On();    // LCD 電圧供給をオンにする。
        R_Config_LCD_Start();    // LCD コントローラを起動します。
        R_Config_RTC_Start();    // リアルタイム・クロックを起動します。

        while(1)
        {
            // TODO：ここにアプリケーション・コードを追加します。
        }
}
```

4.2.67 発振停止検出回路

以下に、スマート・コンフィグレータが発振停止検出回路用として出力する API 関数の一覧を示します。

表 4.73 発振停止検出回路 API 関数

API 関数名	周辺機能	機能概要
R_{Config_OSD}_Create	発振停止検出回路	発振停止検出回路モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_OSD}_Start		発振停止検出回路の動作を開始します。
R_{Config_OSD}_Stop		発振停止検出回路の動作を停止します。
R_{Config_OSD}_Create_UserInit		発振停止検出回路のユーザ独自の初期化処理を実行します。
r_{Config_OSD}_interrupt		INTOSDC 割り込みに伴う処理を実行します。

R_{Config_OSD}_Create

発振停止検出回路モジュールを制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、main() 関数が実行される前に R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_OSD}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_OSD}_Start

発振停止検出回路の動作を開始します。

[指定形式]

```
void    R_{Config_OSD}_Start(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_OSD}_Stop

発振停止検出回路の動作を停止します。

[指定形式]

```
void    R_{Config_OSD}_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_OSD}_Create_UserInit

発振停止検出回路のユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_OSD}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_OSD}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_OSD}_interrupt

INTOSDC 割り込みに伴う処理を実行します。

[指定形式]

CCRL78 ツールチェーンの場合:

```
static void __near    r_{ Config_OSD}_interrupt(void);
```

LLVM ツールチェーンの場合:

```
void    r_{Config_OSD}_interrupt(void);
```

IAR ツールチェーンの場合:

```
__interrupt static void    r_{ Config_OSD}_interrupt(void);
```

[引数]

なし

[戻り値]

なし

使用例

発振停止検出回路の検出時にフラグを設定する例です。

main.c

```
#include "r_smc_entry.h"
extern uint8_t osd_flag;

void main(void);

void main(void)
{
    EI();    // グローバル割り込みを有効にして割り込み処理を許可します。
    R_Config_OSD_Start();    // 発振停止検出回路モジュールを起動します。
    while(1U);    // プログラムを実行し続けるためのループ。
}
```

Config_OSD_user.c

```
/* Start user code for global. Do not edit comment generated here */
uint8_t osd_flag = 0U;
/* End user code. Do not edit comment generated here */

static void __near r_Config_OSD_interrupt(void)
{
    /* Start user code for r_Config_OSD_interrupt. Do not edit comment generated here */
    osd_flag++;    // 発振停止検出時にフラグをインクリメントします。
    /* End user code. Do not edit comment generated here */
}
```

Config_WDT_user.c

```
static void __near r_Config_WDT_interrupt(void)
{
    /* Start user code for r_Config_WDT_interrupt. Do not edit comment generated here */
    R_Config_WDT_Restart();    // システムのリセットを防ぐために、ウォッチドッグ・タイマを再起動します。
    /* End user code. Do not edit comment generated here */
}
```

4.2.68 外部サンプリング

以下に、スマート・コンフィグレータが外部サンプリング用として出力する API 関数の一覧を示します。

表 4.74 外部サンプリング API 関数

API 関数名	周辺機能	機能概要
R_{Config_EXSD}_Create	発振停止検出回路	外部サンプリング・モジュールを制御する前に必要な初期化処理を実行します。
R_{Config_EXSD}_Start		外部サンプリングの動作を開始します。
R_{Config_EXSD}_Stop		外部サンプリングの動作を停止します。
R_{Config_EXSD}_Create_UserInit		外部サンプリングのユーザ独自の初期化処理を実行します。
r_{Config_EXSD}_interrupt		INTEXSD 割り込みに伴う処理を実行します。

R_{Config_EXSD}_Create

外部サンプリング・モジュールを制御する前に必要な初期化処理を実行します。

備考　　この API 関数は、main() 関数が実行される前に R_Systeminit から呼び出されます。

[指定形式]

```
void    R_{Config_EXSD}_Create(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_EXSD}_Start

外部サンプリングの動作を開始します。

[指定形式]

```
void    R_{Config_EXSD}_Start(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_EXSD}_Stop

外部サンプリングの動作を停止します。

[指定形式]

```
void    R_{Config_EXSD}_Stop(void);
```

[引数]

なし

[戻り値]

なし

R_{Config_EXSD}_Create_UserInit

外部サンプリングのユーザ独自の初期化処理を実行します。

備考　この API 関数は、R_{Config_EXSD}_Create のコールバック・ルーチンとして呼び出されます。

[指定形式]

```
void    R_{Config_EXSD}_Create_UserInit(void);
```

[引数]

なし

[戻り値]

なし

r_{Config_EXSD}_interrupt

INTEXSD 割り込みに伴う処理を実行します。

備考　INTEXSD 割り込みは、停止している CPU／周辺ハードウェア・クロック周波数 (fCLK) と同期して動作するため、STOP モードおよび SNOOZE モードでは使用できません。

［指定形式］

CCRL78 ツールチェーンの場合:

```
static void __near    r_{ Config_EXSD}_interrupt(void);
```

LLVM ツールチェーンの場合:

```
void    r_{Config_EXSD}_interrupt(void);
```

IAR ツールチェーンの場合:

```
__interrupt static void    r_{ Config_EXSD}_interrupt(void);
```

［引数］

なし

［戻り値］

なし

使用例

外部サンプリング検出時に信号を出力する例です。

main.c

```
#include "r_smc_entry.h"

void main(void);

void main(void)
{
    EI();   // グローバル割り込みを有効にして割り込み処理を許可します。
    R_Config_EXSD_Start();     // 外部サンプリング・モジュールを起動して、外部サンプリングのモニタリングを開始します
    R_Config_IT00_Start();    // 出力周期を提供するために、8 ビット・インターバル・タイマ 00 を起動します。
    while(1U);
}
```

Appedix A: API 関数の比較

コード生成ツールによって出力される API 関数とスマート・コンフィグレータによって出力される API 関数の比較表です。 コード生成ツールの API 関数に対応するスマート・コンフィグレータの API 関数を理解できます。

表 A.1　　コード生成ツールとスマート・コンフィグレータの API（1/5）

周辺機能	コード生成ツール API 関数	スマート・コンフィグレータ API 関数
共通	main	main
	R_MAIN_UserserInit	–
	hdwinit	–
	R_Systeminit	R_Systeminit
	low_level_init	–
	HardwareSetup	–
クロック発生回路	R_CGC_Create	–
	R_CGC_Set_ClockMode	–
	R_CGC_Create_UserInit	–
	R_CGC_Get_ResetSource	–
ポート機能	R_PORT_Create	R_{Config_PORT}_Create
	R_PORT_Create_UserInit	R_{Config_PORT}_Create_UserInit
タイマ・アレイ・ユニット	R_TAU*m*_Create	R_TAU*m*_Create
	R_TAU*m*_Channel*n*_Start	R_{Config_TAU*m*_*n*}_Start
	R_TAU*m*_Channel*n*_Higher8bits_Start	R_{Config_TAU*m*_*n*}_Higher8bits_Start
	R_TAU*m*_Channel*n*_Lower8bits_Start	R_{Config_TAU*m*_*n*}_Lower8bits_Start
	R_TAU*m*_Channel*n*_Stop	R_{Config_TAU*m*_*n*}_Stop
	R_TAU*m*_Channel*n*_Higher8bits_Stop	R_{Config_TAU*m*_*n*}_Higher8bits_Stop
	R_TAU*m*_Channel*n*_Lower8bits_Stop	R_{Config_TAU*m*_*n*}_Lower8bits_Stop
	R_TAU*m*_Reset	R_TAU*m*_Set_Reset
	R_TAU*m*_Set_PowerOff	R_TAU*m*_Set_PowerOff
	R_TAU*m*_Channel*n*_Get_PulseWidth	R_{Config_TAU*m*_*n*}_Get_PulseWidth
	R_TAU*m*_Channel*n*_Set_SoftwareTriggerOn	R_{Config_TAU*m*_*n*}_Set_SoftwareTriggerOn
	R_TAU*m*_Create_UserInit	R_{Config_TAU*m*_*n*}_Create_UserInit
	r_tau*m*_channel*n*_interrupt	r_{Config_TAU*m*_*n*}_interrupt
	r_tau*m*_channel*n*_higher8bits_interrupt	r_{Config_TAU*m*_*n*}_higher8bits_interrupt
リアルタイム・クロック	R_RTC_Create	R_{Config_RTC}_Create
	R_RTC_Start	R_{Config_RTC}_Start
	R_RTC_Stop	R_{Config_RTC}_Stop
	R_RTC_Set_PowerOff	R_RTC_Set_PowerOff
	R_RTC_Set_HourSystem	R_{Config_RTC}_Set_HourSystem
	R_RTC_Set_CounterValue	R_{Config_RTC}_Set_CounterValue
	R_RTC_Get_CounterValue	R_{Config_RTC}_Get_CounterValue
	R_RTC_Set_ConstPeriodInterruptOn	R_{Config_RTC}_Set_ConstPeriodInterruptOn
	R_RTC_Set_ConstPeriodInterruptOff	R_{Config_RTC}_Set_ConstPeriodInterruptOff
	R_RTC_Set_AlarmOn	R_{Config_RTC}_Set_AlarmOn
	R_RTC_Set_AlarmOff	R_{Config_RTC}_Set_AlarmOff
	R_RTC_Set_AlarmValue	R_{Config_RTC}_Set_AlarmValue
	R_RTC_Get_AlarmValue	R_{Config_RTC}_Get_AlarmValue
	R_RTC_Set_RTC1HZOn	R_{Config_RTC}_Set_RTC1HZOn
	R_RTC_Set_RTC1HZOff	R_{Config_RTC}_Set_RTC1HZOff
	R_RTC_Create_UserInit	R_{Config_RTC}_Create_UserInit
	r_rtc_interrupt	r_{Config_RTC}_interrupt
	r_rtc_callback_constperiod	r_{Config_RTC}_callback_constperiod
	r_rtc_callback_alarm	r_{Config_RTC}_callback_alarm
クロック出力／ブザー出力制御回路	R_PCLBUZ*n*_Create	R_{Config_PCLBUZ*n*}_Create
	R_PCLBUZ*n*_Start	R_{Config_PCLBUZ*n*}_Start
	R_PCLBUZ*n*_Stop	R_{Config_PCLBUZ*n*}_Stop
	R_PCLBUZ*n*_Create_UserInit	R_{Config_PCLBUZ*n*}_Create_UserInit

表 A.2　　コード生成ツールとスマート・コンフィグレータの API（2/5）

周辺機能	コード生成ツール API 関数	スマート・コンフィグレータ API 関数
ウォッチドッグ・タイマ	R_WDT_Create	R_{Config_WDT}_Create
	R_WDT_Restart	R_{Config_WDT}_Restart
	R_WDT_Create_UserInit	R_{Config_WDT}_Create_UserInit
	r_wdt_interrupt	r_{Config_WDT}_interrupt
A/D コンバータ	R_ADC_Create	R_{Config_ADC}_Create
	R_ADC_Start	R_{Config_ADC}_Start
	R_ADC_Stop	R_{Config_ADC}_Stop
	R_ADC_Set_OperationOn	R_{Config_ADC}_Set_OperationOn
	R_ADC_Set_OperationOff	R_{Config_ADC}_Set_OperationOff
	R_ADC_Reset	R_ADC_Set_Reset
	R_ADC_Set_PowerOff	R_ADC_Set_PowerOff
	R_ADC_Set_ADChannel	R_{Config_ADC}_Set_ADChannel
	R_ADC_Set_SnoozeOn	R_{Config_ADC}_Set_SnoozeOn
	R_ADC_Set_SnoozeOff	R_{Config_ADC}_Set_SnoozeOff
	R_ADC_Set_TestChannel	R_{Config_ADC}_Set_TestChannel
	R_ADC_Get_Result	R_{Config_ADC}_Get_Result_10bit
	R_ADC_Get_Result_8bit	R_{Config_ADC}_Get_Result_8bit
	R_ADC_Create_UserInit	R_{Config_ADC}_Create_UserInit
	r_adc_interrupt	r_{Config_ADC}_interrupt
D/A コンバータ	R_DAC_Create	R_DAC_Create
	R_DAC*n*_Start	R_{Config_DAC*n*}_Start
	R_DAC*n*_Stop	R_{Config_DAC*n*}_Stop
	R_DAC_Set_PowerOff	R_DAC_Set_PowerOff
	R_DAC*n*_Set_ConversionValue	R_{Config_DAC*n*}_Set_ConversionValue
	R_DAC_Reset	R_DAC_Set_Reset
	R_DAC*n*_Create_UserInit	R_{Config_DAC*n*}_Create_UserInit
コンパレータ	R_COMP_Create	R_COMP_Create
	R_COMP*n*_Start	R_{Config_COMP*n*}_Start
	R_COMP*n*_Stop	R_{Config_COMP*n*}_Stop
	R_COMP_Reset	R_COMP_Set_Reset
	R_COMP_Set_PowerOff	R_COMP_Set_PowerOff
	R_COMP_Create_UserInit	R_{Config_COMP*n*}_Create_UserInit
	r_comp*n*_interrupt	r_{Config_COMP*n*}_interrupt
プログラマブル・ゲイン・アンプ	R_PGA_Create	R_{Config_PGA}_Create
	R_PGA_Start	R_{Config_PAG}_Start
	R_PGA_Stop	R_{Config_PAG}_Stop
	R_PGA_Create_UserInit	R_{Config_PGA}_Create_UserInit
シリアル・アレイ・ユニット	R_SAU*m*_Create	R_SAU*m*_Create
	R_SAU*m*_Reset	R_SAU*m*_Set_Reset
	R_SAU*m*_Set_PowerOff	R_SAU*m*_Set_PowerOff
	R_SAU*m*_Set_SnoozeOn	R_SAU*m*_Set_SnoozeOn
	R_SAU*m*_Set_SnoozeOff	R_SAU*m*_Set_SnoozeOff
	R_UART*n*_Create	R_{Config_UART*q*}_Create
	R_UART*n*_Start	R_{Config_UART*q*}_Start
	R_UART*n*_Stop	R_{Config_UART*q*}_Stop
	R_UART*n*_Send	R_{Config_UART*q*}_Send
	R_UART*n*_Receive	R_{Config_UART*q*}_Receive
	R_CSI*mn*_Create	R_{Config_CSI*p*}_Create
	R_CSI*mn*_Start	R_{Config_CSI*p*}_Start
	R_CSI*mn*_Stop	R_{Config_CSI*p*}_Stop
	R_CSI*mn*_Send	R_{Config_CSI*p*}_Send
	R_CSI*mn*_Receive	R_{Config_CSI*p*}_Receive
	R_CSI*mn*_Send_Receive	R_{Config_CSI*p*}_Send_Receive
	R_IIC*mn*_Create	R_{Config_IIC*r*}_Create
	R_IIC*mn*_StartCondition	R_{Config_IIC*r*}_StartCondition
	R_IIC*mn*_StopCondition	R_{Config_IIC*r*}_StopCondition
	R_IIC*mn*_Stop	R_{Config_IIC*r*}_Stop
	R_IIC*mn*_Master_Send	R_{Config_IIC*r*}_Master_Send
	R_IIC*mn*_Master_Receive	R_{Config_IIC*r*}_Master_Receive
	R_SAU*m*_Create_UserInit	-

表 A.3　コード生成ツールとスマート・コンフィグレータの API（3/5）

周辺機能	コード生成ツール API 関数	スマート・コンフィグレータ API 関数
シリアル・アレイ・ユニット	r_uart*n*_interrupt_send	r_{Config_UART*q*}_interrupt_send
	r_uart*n*_interrupt_receive	r_{Config_UART*q*}_interrupt_receive
	r_uart*n*_interrupt_error	r_{Config_UART*q*}_interrupt_error
	r_uart*n*_callback_sendend	r_{Config_UART*q*}_callback_sendend
	r_uart*n*_callback_receiveend	r_{Config_UART*q*}_callback_receiveend
	r_uart*n*_callback_error	r_{Config_UART*q*}_callback_error
	r_uart*n*_callback_softwareoverrun	r_{Config_UART*q*}_callback_softwareoverrun
	r_csi*mn*_interrupt	r_{Config_CSI*p*}_interrupt
	r_csi*mn*_callback_sendend	r_{Config_CSI*p*}_callback_sendend
	r_csi*mn*_callback_receiveend	r_{Config_CSI*p*}_callback_receiveend
	r_csi*mn*_callback_error	r_{Config_CSI*p*}_callback_error
	r_iic*mn*_interrupt	r_{Config_IIC*r*}_interrupt
	r_iic*mn*_callback_master_sendend	r_{Config_IIC*r*}_callback_master_sendend
	r_iic*m*n_callback_master_receiveend	r_{Config_IIC*r*}_callback_master_receiveend
	r_iic*mn*_callback_master_error	r_{Config_IIC*r*}_callback_master_error
シリアル・インタフェース IICA	R_IICA*n*_Create	R_{Config_IICA*n*}_Create
	R_IICA*n*_StopCondition	R_{Config_IICA*n*}_StopCondition
	R_IICA*n*_Stop	R_{Config_IICA*n*}_Stop
	R_IICA*n*_Reset	R_IICA*n*_Set_Reset
	R_IICA*n*_Set_PowerOff	R_IICA*n*_Set_PowerOff
	R_IICA*n*_Master_Send	R_{Config_IICA*n*}_Master_Send
	R_IICA*n*_Master_Receive	R_{Config_IICA*n*}_Master_Receive
	R_IICA*n*_Slave_Send	R_{Config_IICA*n*}_Slave_Send
	R_IICA*n*_Slave_Receive	R_{Config_IICA*n*}_Slave_Receive
	R_IICA*n*_Set_SnoozeOn	-
	R_IICA*n*_Set_SnoozeOff	-
	R_IICA*n*_Set_WakeupOn	R_{Config_IICA*n*}_Set_WakeupOn
	R_IICA*n*_Set_WakeupOff	R_{Config_IICA*n*}_Set_WakeupOff
	R_IICA*n*_Create_UserInit	R_{Config_IICA*n*}_Create_UserInit
	r_iica*n*_interrupt	r_{Config_IICA*n*}_interrupt
	r_iica*n*_callback_master_sendend	r_{Config_IICA*n*}_callback_master_sendend
	r_iica*n*_callback_master_receiveend	r_{Config_IICA*n*}_callback_master_receiveend
	r_iica*n*_callback_master_error	r_{Config_IICA*n*}_callback_master_error
	r_iica*n*_callback_slave_sendend	r_{Config_IICA*n*}_callback_slave_sendend
	r_iica*n*_callback_slave_receiveend	r_{Config_IICA*n*}_callback_slave_receiveend
	r_iica*n*_callback_slave_error	r_{Config_IICA*n*}_callback_slave_error
	r_iica*n*_callback_getstopcondition	r_{Config_IICA*n*}_callback_getstopcondition
データ・トランスファ・コントローラ	R_DTC_Create	R_{Config_DTC}_Create
	R_DTC*n*_Start	R_{Config_DTCD*n*}_Start
	R_DTC*n*_Stop	R_{Config_DTCD*n*}_Stop
	R_DTC_Set_PowerOff	R_DTC_Set_PowerOff
	R_DTC_Create_UserInit	R_{Config_DTC}_Create_UserInit
イベント・リンク・コントローラ	R_ELC_Create	R_{Config_ELC}_Create
	R_ELC_Stop	R_{Config_ELC}_Stop
	R_ELC_Create_UserInit	R_{Config_ELC}_Create_UserInit
割り込み機能	R_INTC_Create	R_{Config_INTC}_Create
	R_INTC*n*_Start	R_{Config_INTC}_INTP*n*_Start
	R_INTC*n*_Stop	R_{Config_INTC}_INTP*n*_Stop
	R_INTC_Create_UserInit	R_{Config_INTC}_Create_UserInit
	r_intcn_interrupt	r_{Config_INTC}_intp*n*_interrupt
キー割り込み機能	R_KEY_Create	R_{Config_KR}_Create
	R_KEY_Start	R_{Config_KR}_Start
	R_KEY_Stop	R_{Config_KR}_Stop
	R_KEY_Create_UserInit	R_{Config_KR}_Create_UserInit
	r_key_interrupt	r_{Config_KR}_interrupt
電圧検出回路	R_LVD_Create	R_{Config_LVD*n*}_Create
	R_LVD_InterruptMode_Start	R_LVD_Start_Interrupt
	R_LVD_Create_UserInit	R_{Config_LVD*n*}_Create_UserInit
	r_lvd_interrupt	r_lvd_interrupt

表 A.4　　コード生成ツールとスマート・コンフィグレータの API（4/5）

周辺機能	コード生成ツール API 関数	スマート・コンフィグレータ API 関数
タイマ RD	R_TMRD*n*_Create	R_TRD_Create
	R_TMRD*n*_Start	R_{Config_TRD*n*}_Start
	R_TMRD*n*_Stop	R_{Config_TRD*n*}_Stop
	R_TMRD*n*_Set_PowerOff	R_TRD_Set_PowerOff
	R_TMRD*n*_ForcedOutput_Start	R_TRD_ForcedOutput_Enable
	R_TMRD*n*_ForcedOutput_Stop	R_TRD_ForcedOutput_Disable
	R_TMRD*n*_Get_PulseWidth	R_{Config_TRD*n*}_Get_PulseWidth
	R_TMRD_PWMOP_ForcedOutput_Stop	R_{Config_PWMOPA}_Software_Release
	R_TMRD_PWMOP_Set_PowerOff	R_PWMOPA_Set_PowerOff
	R_TMRD*n*_Create_UserInit	R_{Config_TRD*n*}_Create_UserInit
	r_tmrd*n*_interrupt	r_{Config_TRD*n*}_trd*n*_interrupt
タイマ RJ	R_TMRJ*n*_Create	R_{Config_TRJ*n*}_Create
	R_TMRJ*n*_Create_UserInit	R_{Config_TRJ*n*}_Create_UserInit
	r_tmrj*n*_interrupt	r_{Config_TRJ*n*}_interrupt
	R_TMRJ*n*_Start	R_{Config_TRJ*n*}_Start
	R_TMRJ*n*_Stop	R_{Config_TRJ*n*}_Stop
	R_TMRJ*n*_Set_PowerOff	R_TRJ_Set_PowerOff
	R_TMRJ*n*_Get_PulseWidth	R_{Config_TRJ*n*}_Get_PulseWidth
インターバル・タイマ（12 ビット・インターバル・タイマ）	R_IT_Create	R_{Config_IT}_Create
	R_IT_Create_UserInit	R_{Config_IT_Create_UserInit
	r_it_interrupt	r_{Config_IT}_interrupt
	R_IT_Start	R_{Config_IT}_Start
	R_IT_Stop	R_{Config_IT}_Stop
	R_IT_Set_PowerOff	R_IT_Set_PowerOff
タイマ RG	R_TMRG*n*_Create	R_{Config_TRG}_Create
	R_TMRG*n*_Create_UserInit	R_{Config_TRG}_Create_UserInit
	r_tmrg*n*_interrupt	r_{Config_TRG}_interrupt
	R_TMRG*n*_Start	R_{Config_TRG}_Start
	R_TMRG*n*_Stop	R_{Config_TRG}_Stop
	R_TMRG*n*_Set_PowerOff	R_TRG_Set_PowerOff
	R_TMRG*n*_Get_PulseWidth	R_{Config_TRG}_Get_PulseWidth
タイマ RX	R_TMRX_Create	R_{Config_TRX}_Create
	R_TMRX_Create_UserInit	R_{Config_TRX}_Create_UserInit
	r_tmrx_interrupt	r_{Config_TRX}_interrupt
	R_TMRX_Start	R_{Config_TRX}_Start
	R_TMRX_Stop	R_{Config_TRX}_Stop
	R_TMRX_Set_PowerOff	R_TRX_Set_PowerOff
	R_TMRX_Get_BufferValue	R_{Config_TRX}_Get_BufferValue

表 A.5　　コード生成ツールとスマート・コンフィグレータの API（5/5）

周辺機能	コード生成ツール API 関数	スマート・コンフィグレータ API 関数
タイマ KB	R_KB*n*_Create	R_{Config_TKBn}_Create
	R_KB*n*_Start	R_{Config_TKBn}_Start
	R_KB*n*_Stop	R_{Config_TKBn}_Stop
	R_KB*n*_Simultaneous_Start	-
	R_KB*n*_Simultaneous_Stop	-
	R_KB*n*_Synchronous_Start	-
	R_KB*n*_Synchronous_Stop	-
	R_KB*n*_TKBO*m*0_SmoothStartFunction_Start	R_{Config_TKB*n*}_TKBO*n*0_SmoothStart Function_Start
	R_KB*n*_TKBO*m*0_SmoothStartFunction_Stop	R_{Config_TKB*n*}_TKBO*n*0_SmoothStart Function_Stop
	R_KB*n*_TKBO*m*1_SmoothStartFunction_Start	R_{Config_TKB*n*}_TKBO*n*1_SmoothStart Function_Start
	R_KB*n*_TKBO*m*1_SmoothStartFunction_Stop	R_{Config_TKB*n*}_TKBO*n*1_SmoothStart Function_Stop
	R_KB*n*_Set_BatchOverwriteRequestOn	R_{Config_TKB*n*}_Set_BatchOverwrite RequestOn
	R_KB*n*_TKBO*m*0_Forced_Output_Stop_Function1_Start	R_{Config_TKB*n*}_TKBO*n*0_Forced_ Output_Stop_Function1_Start
	R_KB*n*_TKBO*m*0_Forced_Output_Stop_Function1_Stop	R_{Config_TKB*n*}_TKBO*n*0_Forced_ Output_Stop_Function1_Stop
	R_KB*n*_TKBO*m*1_Forced_Output_Stop_Function1_Start	R_{Config_TKB*n*}_TKBO*n*1_Forced_ Output_Stop_Function1_Start
	R_KB*n*_TKBO*m*1_Forced_Output_Stop_Function1_Stop	R_{Config_TKB*n*}_TKBO*n*1_Forced_ Output_Stop_Function1_Stop
	R_KB*n*_Set_PowerOff	R_TKB_Set_PowerOff
	R_KB*n*_Create_UserInit	R_{Config_TKB*n*}_Create_UserInit
	r_kb*n*_interrupt	r_{Config_TKB*n*}_end_count_interrupt
LCD コントローラ／ドライバ	R_LCD_Create	R_{Config_LCD}_Create
	R_LCD_Create_UserInit	R_{Config_LCD}_Create_UserInit
	R_LCD_Start	R_{Config_LCD}_Start
	R_LCD_Stop	R_{Config_LCD}_Stop
	R_LCD_Voltage_On	R_{Config_LCD}_Voltage_On
	R_LCD_Voltage_Off	R_{Config_LCD}_Voltage_Off
	-	R_{Config_LCD}_Set_DisplayData

改訂記録

Rev.	セクション	ポイント
1.00	-	初版発行
1.01	2.1 説明	r_{Config_UARTAn}_PollingEnd_UserCode()を追加 r_{Config_UARTAn}_send_1byte() と R_{Config_UARTAn}_Send_Polling()を削除
	4.2 関数リファレンス	R_{Config_ELCL_xxx}_Create()の備考を更新 r_{Config_ELCL_xxx}_interrupt()を追加
1.02	全章	コールバック関数の備考 更新
	4.2 関数リファレンス	表 4.1／表 4.2 共通用 API 関数 更新
		4.2.6 外部イベント・カウンタ（タイマ RJ）追加
		4.2.8 入力信号のハイ／ロウ・レベル幅測定（タイマ RJ）追加
		4.2.10 PWM 出力（PWM モードを使用したタイマ RD／拡張 PWM モード）追加
		4.2.11 PWM 出力（PWM モードを使用したタイマ RD／PWM3 モード／拡張 PWM モード）追加
		4.2.12 入力パルス間隔／周期測定（タイマ・アレイ・ユニット）更新
		4.2.13 入力パルス間隔／周期測定（タイマ RJ）追加
		4.2.15 インターバル・タイマ（タイマ RJ）追加
		4.2.18 方形波出力（タイマ RJ）追加
		4.2.22 インプットキャプチャ機能（タイマ RJ）追加
		4.2.23 アウトプットコンペア機能（タイマ RJ）追加
		4.2.24 三相 PWM 出力（タイマ RJ）追加
		4.2.25 PWM オプション・ユニット A（タイマ RJ）追加
		4.2.29 12 ビット A/D シングル・スキャン 追加
		4.2.30 12 ビット A/D 連続スキャン 追加
		4.2.31 12 ビット A/D グループ・スキャン 追加
		4.2.38 UART 通信（LIN/UART モジュール）追加
		4.2.41 I2C 通信(スレーブ・モード)（シリアル・インタフェース IICA）更新 R_{Config_IICA*n*}_Slave_Send, R_{Config_IICA*n*}_Slave_Receive, r_{Config_IICA*n*}_slave_handler
		4.2.49 イベントリンクコントロール 追加
	Appedix A: API 関数の比較	Appedix A 更新

Rev.	セクション	ポイント
1.03	2.1 説　　明	表 2.2 出力ファイル (2/14) 更新
	4.2 関数リファレンス	表 2.6 出力ファイル (6/14) 更新
		表 2.14 出力ファイル (14/14) 更新
		表 4.1 共通用 API 関数（1/3）更新
		表 4.2 共通用 API 関数（2/3）更新
		4.2.1 共通 R_ITL_Start_Interrupt 更新
		4.2.1 共通 R_TRD_ForcedOutput_Enable 追加
		4.2.1 共通 R_TRD_ForcedOutput_disable 追加
		4.2.1 共通 R_IT_Set_PowerOn 追加
		4.2.1 共通 R_IT_Set_PowerOff 追加
		4.2.16 インターバル・タイマ (12 ビット・インターバル・タイマ) 追加
		4.2.17 ワンショット・パルス出力 R_{Config_TAU*m*_*n*}_Get_PulseWidth 追加
		4.2.20 インターバル・タイマ（8 ビット・カウント・モード） R_{Config_ITL*n*}_Start 更新
		4.2.21 インターバル・タイマ（16 ビット・カウント・モード） R_{Config_ITL*n*_ITL*m*}}_Start 更新
		4.2.22 インターバル・タイマ（32 ビット・カウント・モード） R_{Config_ITL000_ITL001_ITL012_ITL013}_Start 更新
		4.2.49 ロジック＆イベント・リンク・コントローラ (ELCL) 更新
	Appedix A: API 関数の比較	表 A.4 コード生成ツールとスマート・コンフィグレータの API（4/4）更新
1.04	2.1 説　　明	表 2.2 出力ファイル (2/14) 更新
	4.2 関数リファレンス	4.2.1 共通　R_TRD_Set_Reset, R_TRD_Release_Reset, R_PWMOPA_Set_Reset, R_PWMOPA_Release_Reset, R_TRJ_Set_Reset, R_TRJ_Release_Reset, R_TRG_Set_PowerOn, R_TRG_Set_PowerOff, R_TRG_Set_Reset, R_TRG_Release_Reset, R_TRX_Set_PowerOn, R_TRX_Set_PowerOff, R_TRX_Set_Reset, R_TRX_Release_Reset, R_TKB_Create, R_TKB_Set_PowerOn, R_TKB_Set_PowerOff, R_TKB_Set_Reset, R_TKB_Release_Reset, R_PGACOMP_Create, R_PGACOMP_Set_PowerOn, R_PGACOMP_Set_PowerOff, R_PGACOMP_Set_Reset, R_PGACOMP_Release_Reset, R_DALI_Set_PowerOn, R_DALI_Set_PowerOff, R_DALI_Set_Reset, R_DALI_Release_Reset　追加
		4.2.7 入力信号のハイ／ロウ・レベル幅測定（タイマ・アレイ・ユニット）更新
		4.2.11 PWM 出力（PWM モードを使用したタイマ RD0 と RD1／PWM3 モード／拡張 PWM モード／タイマ KB3 PWM 出力ゲートモード）にタイマ KB3 追加
		4.2.12 PWM 出力（PWM モードを使用したタイマ RG／PWM2 モード）追加
		4.2.13 PWM 出力（タイマ KB を使用した同時スタート／ストップ・モード（TKBCRn0 レジスタによる周期制御）／同時スタート／ストップ・モード（外部トリガ入力による周期制御）／同時スタート／ストップ・モード（マスタによる周期制御）追加

Rev.	セクション	ポイント
1.04	4.2 関数リファレンス	4.2.14 PWM 出力（タイマ KB を使用した同時スタート／ストップ・モード（TKBCRn0 レジスタによる周期制御）／同時スタート／ストップ・モード（外部トリガ入力による周期制御）／同時スタート／ストップ・モード（マスタによる周期制御））（1 つスレーブ）追加
		4.2.15 PWM 出力（タイマ KB を使用した同時スタート／ストップ・モード（TKBCRn0 レジスタによる周期制御）／同時スタート／ストップ・モード（外部トリガ入力による周期制御）／同時スタート／ストップ・モード（マスタによる周期制御））（2 つスレーブ）追加
		4.2.16 入力パルス間隔／周期測定（タイマ・アレイ・ユニット）備考リンク更新
		4.2.18 インターバル・タイマ（タイマ・アレイ・ユニット）の使用例を更新
		4.2.22 方形波出力（タイマ・アレイ・ユニット）と使用例の更新
		4.2.28 インプットキャプチャ機能（タイマ RG）追加
		4.2.29 インプットキャプチャ機能（タイマ RX）追加
		4.2.31 アウトプットコンペア機能（タイマ RG）追加
		4.2.34 位相計数モード追加
		4.2.37A/D コンバータの API 追加
		4.2.42 データ・トランスファ・コントローラ 備考リンク 更新
		4.2.43 コンパレータ 備考リンク 更新
		4.2.44 プログラマブル・ゲイン・アンプ追加
		4.2.46 UART 通信（シリアル・アレイ・ユニット）説明と使用例 更新
		4.2.47 UART 通信（シリアル・インタフェース UARTA）R_{Config_UARTq}_Send に説明と R_{Config_UARTAn}_Create_UserInit の備考リンクを更新
		4.2.48 UART 通信（LIN/UART モジュール）R_{Config_RLIN3n}_Send に説明と r_{Config_RLIN3n}_callback_receiveend の備考リンクを更新
		4.2.49 DALI 通信（コントロールデバイス）追加
		4.2.50 DALI 通信（コントロールギア）追加
	Appedix A: API 関数の比較	表 A.1, 表 A.4, 表 A.5 更新

Rev.	セクション	ポイント
1.05	2.1 説明	表 2.7 出力ファイル（7/22）:以下の API 削除 R_{Config_TKBn}_TKBOn0_DitheringFunction_Start, R_{Config_TKBn}_TKBOn0_DitheringFunction_Stop, R_{Config_TKBn}_TKBOn1_DitheringFunction_Start, R_{Config_TKBn}_TKBOn1_DitheringFunction_Stop
		表 2.8 出力ファイル（8/22）:以下の API 削除 R_{Config_TKB0_TKBn}_TKBOm0_DitheringFunction_Start, R_{Config_TKB0_TKBn}_TKBOm0_DitheringFunction_Stop, R_{Config_TKB0_TKBn}_TKBOm1_DitheringFunction_Start, R_{Config_TKB0_TKBn}_TKBOm1_DitheringFunction_Stop
		表 2.9 出力ファイル（9/22）:以下の API 削除 R_{Config_TKB0_TKB1_TKB2}_TKBOn0_DitheringFunction_Start, R_{Config_TKB0_TKB1_TKB2}_TKBOn0_DitheringFunction_Stop, R_{Config_TKB0_TKB1_TKB2}_TKBOn1_DitheringFunction_Start, R_{Config_TKB0_TKB1_TKB2}_TKBOn1_DitheringFunction_Stop
	4.2 関数リファレンス	4.2.13 PWM 出力（タイマ KB を使用した単体動作モード（TKBCRn0 レジスタによる周期制御）／単体動作モード（外部トリガ入力による周期制御）／インターリーブ PFC 出力モード）:以下の API 削除 R_{Config_TKBn}_TKBOn0_DitheringFunction_Start, R_{Config_TKBn}_TKBOn0_DitheringFunction_Stop, R_{Config_TKBn}_TKBOn1_DitheringFunction_Start, R_{Config_TKBn}_TKBOn1_DitheringFunction_Stop
		4.2.14 PWM 出力（タイマ KB を使用した同時スタート／ストップ・モード（TKBCRn0 レジスタによる周期制御）／同時スタート／ストップ・モード（外部トリガ入力による周期制御）／同時スタート／ストップ・モード（マスタによる周期制御））（1 つスレーブ）:以下の API 削除 R_{Config_TKB0_TKBn}_TKBOm0_DitheringFunction_Start, R_{Config_TKB0_TKBn}_TKBOm0_DitheringFunction_Stop, R_{Config_TKB0_TKBn}_TKBOm1_DitheringFunction_Start, R_{Config_TKB0_TKBn}_TKBOm1_DitheringFunction_Stop
		4.2.15 PWM 出力（タイマ KB を使用した同時スタート／ストップ・モード（TKBCRn0 レジスタによる周期制御）／同時スタート／ストップ・モード（外部トリガ入力による周期制御）／同時スタート／ストップ・モード（マスタによる周期制御））（2 つスレーブ）:以下の API 削除 R_{Config_TKB0_TKB1_TKB2}_TKBOn0_DitheringFunction_Start, R_{Config_TKB0_TKB1_TKB2}_TKBOn0_DitheringFunction_Stop, R_{Config_TKB0_TKB1_TKB2}_TKBOn1_DitheringFunction_Start, R_{Config_TKB0_TKB1_TKB2}_TKBOn1_DitheringFunction_Stop
		4.2.48 UART 通信（LIN/UART モジュール）チャネル番号 *n* (*n* = 0, 1, 2)の更新
		4.2.55 電圧検出回路 :以下の API 更新 R_{Config_LVD*n*}_Start R_{Config_LVD*n*}_Stop
	Appedix A: API 関数の比較	表 A.5 コード生成ツールとスマート・コンフィグレータの API（5/5）の更新

Rev.	セクション	ポイント
1.06	1. 概　　説	1.3 注意事項 追加
	2.1 説　　明	表 2.4 出力ファイル（4/22） 追加 r_cg_it8bit_common.c, r_cg_it8bit_common.h, r_cg_it8bit.h, r_cg_lcd.h, r_cg_osd_common.c, r_cg_osd_common.h, r_cg_osd.h
		表 2.5 出力ファイル（5/22） 追加 r_cg_exsd_common.c, r_cg_exsd_common.h, r_cg_exsd.h
		表 2.6 出力ファイル（6/22） 追加 PWM 出力（PWM モード (リモコン・キャリア波形) を使用したタイマ・アレイ・ユニット）
		表 2.22 出力ファイル（22/22） 追加
	4.2 関数リファレンス	4.2.1 共通　表 4.4 共通用 API 関数（4/4）追加 R_ITm_Create, R_ITm_Set_PowerOn, R_ITm_Set_PowerOff, R_OSD_Set_PowerOn, R_OSD_Set_PowerOff, R_OSD_Set_Reset, R_OSD_Release_Reset, R_EXSD_Set_PowerOn, R_EXSD_Set_PowerOff, R_EXSD_Set_Reset, R_EXSD_Release_Reset
		4.2.6 外部イベント・カウンタ（タイマ RJ）更新
		4.2.9 PWM 出力（タイマ・アレイ・ユニット）更新
		4.2.10 PWM 出力（PWM モード（リモコン・キャリア波形）を使用したタイマ・ アレイ・ユニット）追加
		4.2.18 入力パルス間隔／周期測定（タイマ RJ）更新
		4.2.20 インターバル・タイマ（タイマ RJ）更新
		4.2.25 インターバル・タイマ（8 ビット・カウント・モードを使用した 8 ビット・インターバル・タイマ）追加
		4.2.26 インターバル・タイマ（16 ビット・カウント・モードを使用した 8 ビット・インターバル・タイマ）追加
		4.2.32 方形波出力（タイマ RJ）更新
		4.2.65 LCD コントローラ／ドライバ　追加
		4.2.66 発振停止検出回路　追加
		4.2.67 外部サンプリング　追加
	Appedix A: API 関数の比較	表 A.5 コード生成ツールとスマート・コンフィグレータの API（6/6） LCD コントローラ／ドライバ　追加
1.07	2.1 説　　明	表 2.19 出力ファイルリスト（19/22） IIC 通信 (マスタ・モード) (シリアル・インタフェース IICA) の新規機能追 IIC 通信 (マスタ・モード、EEPROM 通信) (シリアル・インタフェース IICA) の新規機能追
	4.2 関数リファレンス	全ての使用例にコメント形式の説明を追加
		4.2.55 IIC 通信 (マスタ・モード) (シリアル・インタフェース IICA)　更新
		4.2.56 IIC 通信 (マスタ・モード、EEPROM 通信) (シリアル・インタフェース IICA)　追加

スマート・コンフィグレータ ユーザーズマニュアル
RL78 APIリファレンス編

発行年月日　2021年04月01日　Rev.1.00
2026年1月20日　Rev.1.07

発行　ルネサス エレクトロニクス株式会社
〒135-0061 東京都江東区豊洲3-2-24（豊洲フォレシア）

スマート・コンフィグレータ

ルネサス エレクトロニクス株式会社

R20UT4852JJ0107